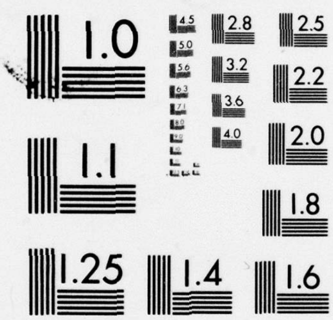
AD-A071 050 GENERAL RESEARCH CORP SANTA BARBARA CA SYSTEMS TECHNO--ETC F/G 9/2
AN EXPERIMENTAL EVALUATION OF SOFTWARE TESTING.(U)
MAY 79 C GANNON, R N MEESON, N B BROOKS F49620-78-C-0103
UNCLASSIFIED GRC-CR-1-854 AFOSR-TR-79-0733 NL

1 OF 2
AD
A071050



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR-TR- 79-0733

LEVEL II

6

B.S.

ADA071050

CR-1-854

An Experimental Evaluation of Software Testing

Final Report

by

C. Gannon

R. N. Meeson

N. B. Brooks

May 1979

SYSTEMS TECHNOLOGIES GROUP

GENERAL RESEARCH CORPORATION

A SUBSIDIARY OF FLOW GENERAL INC.

P.O. Box 6770, Santa Barbara, California 93111

Sponsored by

Air Force Office of Scientific Research

Bolling Air Force Base

Washington, D.C.

Under Contract F49620-78-C-0103

In addition to approval by the Project Leader and Department Head, General Research Corporation reports are subject to independent review by a staff member not connected with the project. This report was reviewed by T. Plambeck.

Sponsored by

Air Force Office of Scientific Research
Bolling Air Force Base
Washington, D.C.
Under Contract F49620-78-C-0103

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) Unclassified

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (18) AFOSR-TR-79-0733
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: (14) GRC-CR-1-854
4. TITLE (and Subtitle) (6) AN EXPERIMENTAL EVALUATION OF SOFTWARE TESTING. (9)
5. TYPE OF REPORT & PERIOD COVERED: (9) FINAL rept.
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) C./GANNON, R. N./Meeson, N. B./Brooks
8. CONTRACT OR GRANT NUMBER(s): (15) F49620-78-C-0103 New
9. PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation, P.O. Box 6770, Santa Barbara, CA 93111
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (16) 61102F 2304 A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, D.C. 20332
12. REPORT DATE: (11) May 1979 (17)
13. NUMBER OF PAGES: 106
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): (12) 107p.
15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Path Testing, C overage, Static Analysis, Tracing, Branch, Test Tool.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the procedures and results of a series of controlled experiments desinged to gather data on actual test tool usage. The primary goal of the experiments was to evaluate the testing techniques of path (branch) coverage testing and static analysis. The evaluation was based on the types of errors detected by these techniques and on a comparison of performance with respect to classical techniques of debug printout and execution tracing. A test program was seeded with errors for the experiments. The error-seeding process is described in detail. continued

DD FORM 1 JAN 73 1473

UNCLASSIFIED 411 275

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (continued)

To date, in spite of much speculation, no computer-aided testing techniques for software have been evaluated in a controlled testing environment. This report discusses and presents the results of a series of such tests.

The techniques evaluated are path (branch) coverage testing and static analysis. The basic approach was to prepare programs for testing by seeding them with errors whose type and frequency are typical of new software at the integration- or system-level of testing.

The experiments were conducted in three phases. The first used eight small programs from a popular programming manual, the second and third used a 5000-line FORTRAN program used to simulate ballistic-missile defense engagements. For the most part, both the path testing and static analysis used the SQLAB tool, with the techniques used singly and in combination. In Phase 1, the DAVE system's static analysis capabilities were also used. In Phase 3, the techniques were compared with the techniques of intermediate-value printout and control-flow tracing.

Of the two techniques, path testing was more effective overall. Its lack of localized error messages was a drawback, but the enhancement to the inspection process was significant, doubling the usual inspection yield. Static analysis, while not as powerful, at times detected errors path testing did not find. It is economical, and its diagnostic message at the error's statement location is a distinct advantage.

The inescapable conclusion remains, however, that fully automated computer-aided testing is not possible at present. Further, the errors that are not detected are generally considered difficult to locate by conventional techniques. In particular, the missing ingredient seems to be a specification of the legal path sequences which a program should be allowed to travel. The error-seeding process is recommended as a measure of testing thoroughness.

CR-1-854

AN EXPERIMENTAL EVALUATION OF SOFTWARE TESTING

Final Report

By

C. Gannon
R.N. Meeson
N.B. Brooks

MAY 1979

ABSTRACT

To date, in spite of much speculation, no computer-aided testing techniques for software have been evaluated in a controlled testing environment. This report discusses and presents the results of a series of such tests.

The techniques evaluated are path (branch) coverage testing and static analysis. The basic approach was to prepare programs for testing by seeding them with errors whose type and frequency are typical of new software at the integration- or system-level of testing.

The experiments were conducted in three phases. The first used eight small programs from a popular programming manual, the second and third used a 5000-line FORTRAN program used to simulate ballistic-missile defense engagements. For the most part, both the path testing and static analysis used the SQLAB tool, with the techniques used singly and in combination. In Phase 1, the DAVE system's static analysis capabilities were also used. In Phase 3, the techniques were compared with the techniques of intermediate-value printout and control-flow tracing.

Of the two techniques, path testing was more effective overall. Its lack of localized error messages was a drawback, but the enhancement to the inspection process was significant, doubling the usual inspection yield. Static analysis, while not as powerful, at times detected errors path testing did not find. It is economical, and its diagnostic message at the error's statement location is a distinct advantage.

The inescapable conclusion remains, however, that fully automated computer-aided testing is not possible at present. Further, the errors that are not detected are generally considered difficult to locate by conventional techniques. In particular, the missing ingredient seems to

be a specification of the legal path sequences which a program should be allowed to travel. The error-seeding process is recommended as a measure of testing thoroughness.

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification

By
Distribution/
Availability Codes

Dist	Avail and/or special
A	

CONTENTS

CONTENTS (cont.)

ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENT

This project was sponsored by the Air Force Office of Scientific Research, Contract Number, F49620-78-C-0103, under program management of Lt. Col. George W. McKemie. Additional research has been encouraged resulting in a follow-on effort that applies adaptive search techniques to software testing.

Participants in the project were C. Gannon, principal investigator, D. Andrews, J. P. Benson, N. B. Brooks, R. N. Meeson, and S. H. Saib.

We also appreciate the interest in this project of J. B. Goodenough of SofTech Inc., F. S. LaMonica of Rome Air Development Center, and E. F. Miller, Jr. of Software Research Associates.

1 INTRODUCTION

This report describes the procedures and results of a series of controlled experiments designed to gather data on actual test tool usage. The primary goal of these experiments was to evaluate and compare two automated testing techniques, path (branch) coverage testing and static analysis, by determining the types of errors each is capable of locating and measuring the computer and engineering time the techniques require to detect each type of error.

An additional goal of the experiment was to observe and compare the relative testing effectiveness in a multi-error environment of a test tool capable of both path testing and static analysis and a sophisticated compiler having automated intermediate value printout and execution tracing features.

The experiments were successful in providing data on error detection rates and level-of-effort required for finding specific types of errors. They also provided a background for analyzing parallel testing strategies in which the human element, as well as the testing tool technique, plays a significant role in the software testing effort. One of the most important byproducts of the error-seeding activity was to indicate the acute vulnerability of software, especially to errors which can mask each other or which never appear for any but the most exhaustive test data.

1.1 BACKGROUND

Histories of several large software development projects have shown that roughly half the cost of bringing such a project to operational capacity is incurred in "testing" the software after the developer (or) the schedule) had declared the product completed.[1,2] In general, this type of testing is intended to demonstrate that the software is ready for operational use; in fact, a large portion of such testing is devoted to detecting and correcting errors that have gone undetected during development. To assist in this difficult process of testing, various computer-aided techniques have been devised and the necessary supporting tools developed. The value of such computer-aided testing techniques has been both challenged and supported extensively.[3] In the few published studies on the subject that reported the use of test tools, there is disagreement on their effectiveness. In none of the studies on medium- or large-scale software, however, have the evaluations been made in a controlled testing environment in which automated tools were actually used. The goal of this project was to run a series of controlled experiments to gather data based on actual test tool usage.

Goodenough[4] states that 40-92 percent of errors could be found using path-testing techniques. He stresses that the limitations of path testing have not been adequately described and that a false sense of confidence of program correctness may develop if only path-testing methods are used. However, Goodenough's view of path testing excludes the functionality of the data, thereby limiting the testing process to structural path execution. We stress that path testing is not intended to be performed without respect paid to the "reasonableness" of the input data.

---

[1] B. W. Boehm, "Software Engineering: R & D Trends and Defense Needs," Proceedings of the Conference on Research Directions in Software Technology, October 1977.

[2] D. S. Alberts, "Economics of Software Quality Assurance," AFIPS Conference Proceedings, Vol. 45, National Computer Conference, 1976.

[3] D. J. Reiffer and R. L. Ettenger, "Test Tools: Are They a Cure-All?" Proceedings of the 1975 Annual Reliability and Maintenance Symposium, IEEE 75CH0918-3ROC, January 1975.

[4] J. B. Goodenough, "A Survey of Program Testing Issues," Proceedings of the Conference on Research Directions in Software Technology October 1977.

The few studies that report quantitative results for analyzing the effectiveness of path testing are in disagreement. Hetzel[1] states that path testing is of "little value" in the detection of errors. Gannon[2] states that systematic functional and structural testing using a well-defined test plan and a path-testing tool produced an error rate of 0.3% after acceptance test for the large JOVIAL program. The disagreement of errors found by path testing is further shown in Table 1.1.

While Mangold states that 92% of the errors in a program might be found, Howden and Goodenough state that perhaps 50% might be found. The word "might" is used because, except for Gannon's work, no path-testing tool was used to obtain the quoted figures. This lack of results has lead to the widely divergent opinions on the value of path testing.

TABLE 1.1

THEORETICAL RESULTS OF PATH TESTING

Total Errors	Path (branch) Testing Detects	% Detected	Source
22	7-14	40-65	Howden[*]
28	6	21	Howden[†]
224	206	92	Mangold[§]
?	?	50	Goodenough[¶]

[*] W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," Computer Science Technical Report II, University of California, San Diego, May 1976.

[†] W. E. Howden, "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978.

[§] E. R. Mangold, Software Error Analysis and Software Policy Implications," IEEE EASCON, 1974, pp. 123-127.

[¶] Goodenough, op.cit.

[1] W. C. Hetzel, An Experimental Analysis of Program Verification Methods, Thesis, University of North Carolina, Chapel Hill, N. C., 1976.

[2] C. Gannon, "A Verification Case Study," Proceedings of AIAA Computers in Aerospace Conference, Los Angeles, November 1977.

Howden's[1] results are based on the analysis of errors in very small programs (fewer than 30 statements). These programs, taken from Kernighan and Plauger,[2] contain examples of common programming blunders and provide a common basis for comparison. Howden, however, did not use a test tool for his analysis. Hence, for the first phase of our testing experiments we subjected these programs to actual path testing and static analysis. A few of these programs were written in PL/1 and had to be translated into FORTRAN so that test tools could be used.

Very early in the experiments, we found that "error" is a very ambiguous concept. In any software system, designers and programmers take certain liberties based on the generality of the program, the programming language and operating system used, and the requirements for meeting size and speed limitations. In an environment that tries to enforce very strict coding standards, ambiguous comments and intentional mixed mode might be called errors. For our purpose, we defined an error as any construct that (1) appeared to violate the program's specification, or (2) relied on nonstandard characteristics of a compiler, operating system, or computer.

1.2 PURPOSE OF EXPERIMENTS

Two software testing techniques, static analysis and dynamic path (branch) testing,[3] are currently receiving a great deal of attention in the world of software engineering. However, empirical evidence of their ability to detect errors is very limited, as is data concerning the resource investment their use requires. Researchers have estimated or intuitively graded these testing methods, as well as such other techniques as interface consistency, symbolic testing, and special values testing.

---

[1] Howden, 1976, op.cit.

[2] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974.

[3] R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, April 1978

This project seeks (1) to demonstrate _empirically_ the types of errors one can expect to uncover, (2) to measure the engineering and computer time which may be required by the two testing techniques for each class of errors, (3) to analyze the relative merits of a test tool containing both testing capabilities and a compiler containing automated, intermediate-value and trace capabilities, and (4) to direct attention to near-term tool enhancements, based on the experimental evidence.

The experiments for this project were conducted in three phases. The first phase examined the small programs from Kernighan and Plauger using the static analysis and path testing capabilities of SQLAB[1] separately and the static analysis capabilities of the DAVE[2] system. These experiments were performed as a preliminary analysis of the two testing techniques. The second phase of experiments was conducted to determine the types of errors that static analysis and path testing are capable of detecting during system-level testing. The experiments involved seeding one error at a time into a medium-sized program and then recording the detection rate and the resources required by each error detection method. The third phase of experiments was designed to evaluate the effectiveness of static analysis and path testing in a multi-error environment. In this phase the two testing techniques are compared with the classical techniques of intermediate value printout and execution tracing automated by a sophisticated compiler. The complete set of experiments is summarized in Table 1.2.

---

[1] D. M. Andrews and J. P. Benson, _Software Quality Laboratory User's Manual_, General Research Corporation CR-4-770, May 1978.

[2] L. D. Fosdick and C. Miesse, _The DAVE System User's Manual_, University of Colorado, CU-CS-106-77, March 1977.

TABLE 1.2

SET OF EXPERIMENTS

Phase	Purpose	Test Object	Test Technique(Tool)
1	Preliminary analysis: Comparison of empirical results with published theoretical results	Eight small programs from *The Elements of Programming Style*	Path testing (SQLAB) Static analysis (SQLAB) Static analysis (DAVE)
2	Determination of types of errors which can be found (single-error experiment)	5000-line trajectory analysis FORTRAN program	Path testing (SQLAB) Static analysis (SQLAB)
3	Evaluation of a test tool in a multi-error experiment	5000-line trajectory analysis FORTRAN program	Path testing and Static analysis (SQLAB) debugging/trace comiler (CDC FTNX)

1.2.1 Description of Path Testing

Path testing is based upon the assumption that executing all the paths in a program is sufficient to reveal a large fraction of the errors when the program is executed. Or, stated another way, paths which have never been tested may harbor errors. The only practical way to systematically check the execution of each path is by using an automated path-testing tool.

The first step in path testing is to develop a graph model of the program using the tool to identify all the paths through it.
This graph model is composed of an input node which represents all entry points to the program, an output node which represents all possible termination or exit points from the program, and a set of nodes which represent all the possible branching points in the program. The nodes are connected by links which represent statements in the program which are executed sequentially between any two branch points. Note that this model assumes that the destination of all branch points in the program can be determined statically. That is, dynamic definition of branch

points (as in FORTRAN-assigned GOTO statements when the statement label list is not included) is not allowed by this model.

In general, it is impractical and unnecessary to test all possible paths through a program. The number of paths increases drastically with the number of branches and loops it contains. For this reason, the criterion of testing all paths through the program is relaxed and replaced by the requirement to exercise all of the links (or segments) in the program graph. These links correspond to all the straight line code executed in the program between branch points and are called "segments" or "decision-to-decision paths" (DD-paths). Note that by relaxing the testing-all-paths criterion to the testing-all-segments criterion, we implicitly assume the sequential independence of segments. However, experience has shown that the order of segments is important, thus emphasizing one aspect of the path-testing methodology: input data must reflect the functional requirement in order to execute the paths in their intended order.

Path segment testing (known in this report as "path testing," and having the same meaning as "branch" or "segment" testing) is usually accomplished in the following manner. A set of test data that results in correct execution of the program is taken as the basic test case. Using this test case, the program is executed and measurements are taken of the number of path segments executed by the basic test data. The data values in the basic test data which have an effect upon the decision (branch) points in the program are then altered so that every path segment is exercised by the set of test data developed in this manner; the program output is examined for errors, and any execution-time errors are recorded. This process is extremely dependent upon the ability of the tester, aided by the test tool, to derive data input values which result in all path segments being executed.

1.2.2 Description of Static Testing

Although, in its current state of development, static analysis is not able to demonstrate the functional correctness of a program it is easy to use and can detect a number of program errors. The static analysis capabilities of the testing tool are:

1. Set/use checking - warning of local variable usage without prior setting or local variable setting with no subsequent usage.
2. Call checking - the number and type of actual parameters for each invocation are checked against the number and type of formal parameters.
3. Mode checking - the left and right side of assignment statements are analyzed for type consistency.
4. Graph checking - the control flow graph is analyzed for structurally unreachable code and loops in which the control variable is not changed.

Even small programs can contain errors not easily visible in the source listing. Figures 1.1 and 1.2 show a sample program listing and static analysis report generated by SQLAB. Except for set/use checking, the error and warning messages appear at the appropriate source statement. Error location definition is an advantage which path testing does not have.

1.3 MAJOR CONCLUSIONS

The set of experiments provided evidence for assessing the effectiveness of separately using two automated testing techniques for detecting errors of the following categories: computational, logic, input/output, data handling, interface, data definition and database. Also provided by the experiments were the amounts of engineering and

```
STATEMENT LISTING
                                                     SUBROUTINE BSORT ( NUM, ARRAY )
NO. LEVEL   LABEL   STATEMENT TEXT...                                                   DDPATHS
-----------------------------------------------------------------------------------------------
  1                 SUBROUTINE BSORT ( NUM, ARRAY )                                     (  1)
  2                 INTEGER ARRAY ( 100 )
  3                 INTEGER SMALL
  4                 INPUT ( / I / NUM )
  5                 IF ( NUM .GT. MAXNUM )                                              (  2-  3)
  6 ( 1)            .  N = MAXNUM
  7 ( 1)            .  CALL ERROR ( NUM )
  8                 ELSE
  9 ( 1)            .  N = NUM
 10                 ENDIF
 11                 I = 2
 12                 WHILE ( I .LE. N )                                                  (  4-  5)
 13 ( 1)            .  IF ( ARRAY ( I - 1 ) .LE. ARRAY ( I ) )                          (  6-  7)
 14 ( 2)            .  .  I = I + 1
 15 ( 1)            .  ELSE
 16 ( 2)            .  .  SMALL = ARRAY ( I )
 17 ( 2)            .  .  ARRAY ( I ) = ARRAY ( I - 1 )
 18 ( 2)            .  .  J = I - 2
 19 ( 2)            .  .  NEXIT = 0
 20 ( 2)            .  .  WHILE ( NEXIT .EQ. 0 )                                        (  8-  9)
 21 ( 3)            .  .  .  IF ( J .GE. 1 )                                            ( 10- 11)
 22 ( 4)            .  .  .  .  IF ( SMALL .LT. ARRAY ( J ) )                           ( 12- 13)
 23 ( 5)            .  .  .  .  .  ARRAY ( J + 1 ) = ARRAY ( J )
 24 ( 5)            .  .  .  .  .  J = J - 1
 25 ( 4)            .  .  .  .  ELSE
 26 ( 5)            .  .  .  .  .  NEXIT = 2
 27 ( 4)            .  .  .  .  ENDIF
 28 ( 3)            .  .  .  ELSE
 29 ( 4)            .  .  .  .  NEXIT = 1
 30 ( 3)            .  .  .  ENDIF
 31 ( 2)            .  .  ENDWHILE
 32 ( 2)            .  .  ARRAY ( J + 1 ) = SMALL
 33 ( 2)            .  .  I = I + 1
 34 ( 1)            .  ENDIF
 35                 ENDWHILE
 36                 CALL PRNT ( N, ARRAY )
 37                 OUTPUT ( / I / NUM, ( ARRAY ( I ), I = 1, NUM ) )
 38                 RETURN
 39                 IFLAG = .TRUE.
 40                 END
-----------------------------------------------------------------------------------------------
```

Figure 1.1. Sample Program Listing from SQLAB

```
STATIC ANALYSIS                              SUBROUTINE BSORT ( NUM , ARRAY )
 STMT     IDENT.LINE  SOURCE...
-------------------------------------------------------------------------------
   7                       CALL ERROR ( NUM )
                      ---------------------------------------------------------
                      -                            CALL ERROR                 -
                      -         ERROR     CALLED WITH  1 ACTUALLY HAS  2 ARGUMENTS  -
                      ---------------------------------------------------------
  39                       IFLAG = .TRUE.
                      ---------------------------------------------------------
                      -                         GRAPH WARNING                 -
                      -    STATEMENT  39  IS UNREACHABLE OR IS IN AN INFINITE LOOP  -
                      ---------------------------------------------------------
                      ---------------------------------------------------------
                      -                          MODE WARNING                 -
                      -  LEFT HAND SIDE HAS MODE INTEGER   RIGHT HAND SIDE HAS MODE LOGICAL  -
                      ---------------------------------------------------------
-------------------------------------------------------------------------------
                      STATEMENT ANALYSIS SUMMARY        ERRORS  WARNINGS
                      --------------------------        ------  --------
                      GRAPH CHECKING                         0         1
                      CALL CHECKING                          1         0
                      MODE CHECKING                          0         1

  NAME       SCOPE          TYPE      MODE        USE       OTHER INFORMATION...
-------------------------------------------------------------------------------
IFLAG       LOCAL         VARIABLE    INTEGER     OUTPUT
                      ---------------------------------------------------------
                      -                       SET/USE WARNING                 -
                      -  VARIABLE IFLAG       SET BUT NEVER USED        REFER TO STATEMENT(S)-
                      -  39
                      ---------------------------------------------------------
MAXNUM      LOCAL         VARIABLE    INTEGER     INPUT
                      ---------------------------------------------------------
                      -                       SET/USE ERROR                   -
                      -  VARIABLE MAXNUM      USED BUT NEVER SET        REFER TO STATEMENT(S)-
                      -    5    6
                      ---------------------------------------------------------
-------------------------------------------------------------------------------
                      SYMBOL ANALYSIS SUMMARY           ERRORS  WARNINGS
                      -----------------------           ------  --------
                      SET/USE CHECKING                       1         1

          THE FOLLOWING NONLOCAL VARIABLES ARE SET...
          ARRAY
```

Figure 1.2. Sample Static Analysis Report from SQLAB.

computer time expended. Table 1.3 summarizes the rate of error detection and resources. Detailed results, including detection rates for each type of error within each category, are provided in Sec. 5. The computer program used as a test object for most of the experiments is described in Sec. 3, and each error type and frequency used in the experiments is described in Sec. 4.

As Table 1.3 indicates, and Sec. 5 describes more fully, path testing is more effective than static analysis at detecting and locating computational, logic, and database errors. Even so, the rate of detection and amount of engineering time required by path testing show it is

TABLE 1.3

SUMMARY OF ERROR CATEGORY DETECTION

Error Category	Detection Rate (%) Static Analysis	Detection Rate (%) Path Testing†	Resources E/C* Static Analysis	Resources E/C* Path Testing†
Computational	14	58		4.0/12.7
Logic	14	63		3.5/11.7
Input/Output	17	17		1.0/14.7
Data Handling	28	28		2.5/7.0
Interface	25	25		4.0/19.5
Data Definition	25	25		1.0/5.0
Data Base	0	38		2.0/13.9
Total	16%	45%	2.0/24.0§	3.1/11.9¶

*E = engineering hours (average per error category).
C = CDC 7600 computer seconds (average per error category).
As a baseline, complete compilation and execution took 10 seconds.

†Path testing combined with inspection aided by path testing.

§All 49 errors seeded simultaneously.

¶Average of all errors detected by path testing.

not sufficient for use as the sole program verification or error detection technique, and it is rather time-consuming. Static analysis requires much less engineering and computer time (per error), but the payoff in finding errors of a system-level nature is not as great.

The multiple-error experiment indicated that an automated means of printing intermediate results and tracing program execution is more effective for locating errors than the combination of path coverage testing and static analysis. The data gathered in this experiment are presented in Sec. 6. The conclusion drawn from the analysis of this data is that redundant functional information embedded in programs is necessary for automated tools to be more effective.

An important outcome of the error-seeding activity was that when program verification is based on demonstrating complete path coverage, one can still expect approximately 25 percent of the program errors to remain. Path testing depends upon some manifestation of an error in the program output. We found that, when known errors were inserted, and the program was executed with complete coverage data derived from path testing, 25 percent of the errors did not cause any change in the output. These errors were not used in the experiments.

It is possible that many of those errors are harmless in one specific application of a general purpose program (e.g., incorrect computations are never used or are corrected before harm is done). It is more likely, however, that the data generated to satisfy the path testing requirements of a specified percentage of coverage cause control flow to execute <u>sequences</u> of paths which do not exhibit the errors. This is one reason why path testing should <u>always</u> be coupled with stress or boundary condition testing. Overall path coverage may not be increased, but the <u>right</u> sequence of paths may be executed to expose errors.

This set of experiments reinforced the intuitive feeling that error detection is a difficult and highly individual process. Even armed with test tools, complete software verification is still very much a function of human intuition and resourcefulness. The software testing process should not depend entirely on any single current state-of-the-art technique but should encompass as many tools as is practical. Attempting to detect seeded errors of specified type and frequency during system-level or acceptance testing provides a valid measure of test data thoroughness (e.g., did the execution output show the presence of the seeded error?) and fault tolerance of the software (e.g., did other parts of the software correct the error?)

It appears that, until software specification and implementation through a computer language are more integrated and standardized, software testing will never be an automated process.

2 PRELIMINARY ANALYSIS

Eight small programs from The Elements of Programming Style[1] were tested using the static and path testing capabilities of SQLAB and the static analysis capability of the DAVE system. These programs, all under 30 source lines, performed such functions as table lookup, binary search, and computing electrical current. Listings of these programs are included in Appendix A.

There were two motives for spending any time at all on these small programs: curiosity, and the fact that Goodenough and Howden both have based comments regarding the validity of path testing on these programs. Neither researcher, however, used an actual path testing tool in making their judgements. Table 2.1 presents our results from analyzing the eight programs. We have categorized the errors found into three levels. We consider the first level errors the most serious in terms of their impact on computed results and possible cost (in a non-test-tool environment) to detect. The third level errors are the least serious.

For these same programs, Howden said that 40-65 percent of the errors might (he did not actually use a tool) be found using path testing. Our experience was that 70 percent of the errors were found by path testing. When the programs were subjected to both static analysis and path testing, 38 percent of the errors were detected by static analysis and another (no overlapping errors considered by the path tester) 38 percent were found by path testing. The errors are further described in Tables 2.2 - 2.4.

---

[1] Kernighan and Plauger, op. cit.

TABLE 2.1

ERROR CLASSIFICATION AND DETECTION RESULTS FOR PROGRAMS FROM THE ELEMENTS OF PROGRAMMING STYLE

CATEGORIES	NUMBER OF ERRORS: Static and Path Testing Combined			Path Testing Alone		
	S	P	X	P	X	
Level 1.						S → static analysis
Uninitialized variables	7	0	0	7	0	P → path testing
Computational logic	0	2	1	2	1	
Loop Logic	0	5	1	5	1	X → undetected error
Level 2.						
Unchecked array boundary	0	0	1	0	1	
Equality comparison	0	1	1	1	1	
Level 3.						
Improper termination	1	1	0	2	0	
Mixed mode	1	1	2	1	3	
Unused variables	1	0	0	0	1	
totals	10	10	6	18	8	
Total Errors = 26						
percent	38%	38%	24%	70%	30%	

In this small exercise, path testing alone uncovered most of the errors found by static analysis. However, errors detected by static analysis used but a fraction of the resources path testing required. In addition, the static analyzer points out errors explicitly. The DAVE system detected all the errors SQLAB did, with the exception of one mixed mode and one improper program termination error. In path testing, the execution output must be studied for possible errors, and the execution coverage reports must be reviewed to determine what paths were taken when erroneous behavior was exhibited. If there is little output produced in a program, then the tester may have to add printout statements to display intermediate results as paths are executed.

TABLE 2.2

STATIC ANALYSIS FOLLOWED BY PATH TESTING
(Errors detected by Static Analysis were removed from consideration before Path Testing)

Elements of Programming Style Programs

Error Types	SINEFCN	CURRENT	NUMALPH	BALANCE	BINSRCH	INTEGR8	FLOATPT	AREATRY
Uninitialized variable	S	S	S					
Incorrect computational logic	* P							
Incorrect loop exit	P			P				
Mixed mode	S	*	P					
Input type mode		*						
Failure to reinitialize in loop		P						
Extra pass through loop				*		P		
Incorrect variable names					SSSS			
Unchecked array bounds					*			
Logical infinite loop					P			
Convergence logic error					P			
Unused array					S			
Equality comparison							*	P
Improper program termination				S			P	

Static Analysis followed by Path Testing
(Errors detected by Static Analysis removed from
Consideration before Path Testing)

for each error:

S → found by static analysis

P → found by path testing

* → not found

TABLE 2.3

PATH TESTING ALONE

Elements of Programming Style Programs

Error Types	SINEFCN	CURRENT	NUMALPH	BALANCE	BINSRCH	INTEGR8	FLOATPT	AREATRY
Uninitialized variable	P	P	P					
Incorrect computational logic	* P							
Incorrect loop exit	P			P				
Mixed mode	*	*	P					
Input type mode		*						
Failure to reinitialize in loop		P						
Extra pass through loop				*		P		
Incorrect variable names					PPPP			
Unchecked array bounds					*			
Logical infinite loop					P			
Convergence logic error					P			
Unused array					P			
Equality comparison							*	P
Improper program termination				P			P	

Path Testing Alone

for each error:

P → found by path testing

* → not found

TABLE 2.4

DAVE SYSTEM TESTING ALONE

<u>Elements of Programming Style</u> Programs

Error Types	SINEFCN	CURRENT	NUMALPH	BALANCE	BINSRCH	INTEGR8	FLOATPT	AREATRY
Uninitialized variable	D	D	D					
Incorrect computational logic	* *							
Incorrect loop exit	*			*				
Mixed mode	*	*	*					
Input type mode		*						
Failure to reinitialize in loop		*						
Extra pass through loop				*		*		
Incorrect variable names					DDDD			
Unchecked array bounds					*			
Logical infinite loop					*			
Convergence logic error					*			
Unused array					D			
Equality comparison							*	*
Improper program termination				*			*	

DAVE System Testing Alone

for each error:

D → found by DAVE

* → not found

3 TEST OBJECT

The program selected for Phases 2 and 3 as the test object for error seeding is an example program from the TRAID subroutine package.[1] TRAID, a GRC software product developed in 1968 to help solve missile trajectory problems, contains 105 modules primarily for calculating powered and guided-flight trajectories and Keplerian orbits. It also includes support routines for vector and matrix operations, conversion of units of measure, plotting, and report generation. TRAID has been in continuous use at GRC since 1968 and has required very few changes or modifications over this period.

The test program computes the closest approach between an ICBM and an interceptor missile. Data for the program includes descriptions of the ICBM's trajectory and the interceptor's flight characteristics, (i.e., thrust, mass, burn time, drag, etc.) and a schedule of interceptor maneuvers.

The test program employs 57 TRAID routines which expand to approximately 5000 lines (over 3000 complete statements) of FORTRAN code. This program was selected for error-seeding because it is stable, believed to be bug-free, and large enough to constitute a realistic debugging problem.

3.1 MODIFICATION OF THE TEST OBJECT

A number of modifications were made to the test program to replace some of the non-ANS-standard FORTRAN code which the SQLAB test tools would not accept, correct errors found during static program testing, and enable the program to process multiple test cases in a single run.

---

[1] T. Plambeck, The Compleat Traidsman, General Research Corporation, IM-711/2, September 1969.

3.1.1 ANS Standard Corrections

A lenient compiler and unenforced coding standards contributed to approximately 167 lines of non-standard FORTRAN code which could not be recognized by the SQLAB test tool. Three types of illegal code had to be corrected: multiple assignment statements, multiple statements per line, and an alien form of DATA statement. Functionally identical ANS-standard FORTRAN code was substituted for the offending statements.

3.1.2 Static Analysis

Static analysis of the unseeded test program using SQLAB revealed several potential sources of error. For example, in one case two locally declared arrays were assumed to occupy contiguous storage space. The second array was used as an overflow area when the first array was filled. Data could be read into the second array but was only referenced by over-subscripting the first array. This error was indicated by SQLAB's set/use checking facility since the contents of the second array were set but never used.

Other errors included incorrect array dimensions and a number of mode violations for data types involving character (Hollerith) data. None of the errors found, however, appeared to have any consequences either to the operation of the program or to the printed results for the example test data set. "Error" as is used here means a violation of the language definition or a dependency on the non-standard characteristics of a particular compiler, operating system, or machine.

3.1.3 Multiple Test Cases

The test program was further modified to enable the processing of multiple test cases in a single run. The main program and two of the TRAID routines were adapted for this purpose. The multiple test case capability was originally intended to simplify the testing process. An added advantage is that a significant portion of TRAID's data manipulation facilities are now exercised by the test program.

3.2 Expanded Data Set

The original test data set taken from the TRAID user's manual exercised 50 percent of the total paths in the test program. SQLAB's instrumentation facilities and the trace file analysis program were used to create additional test cases to increase the number of paths traversed. Based upon module function, size, position in the module hierarchy, and path coverage from initial data, six modules were selected as retesting targets. The expanded test data set resulted from using path testing techniques to modify the initial data set. Using the expanded test data, path coverage for the six modules rose from 44 percent (with initial test data) to 75 percent.

3.2.1 Instrumentation Techniques

Instrumenting a test program using SQLAB causes software probes to be inserted in the program to trace its execution. The program is then run with a test data set and a trace file is produced. The trace file is automatically analyzed and a path coverage report is printed for each module, as shown in Fig. 3.1. Program paths which have not been exercised by the test data are flagged in this report. It is then up to the tester to determine the conditions that cause these paths to be traversed and to devise appropriate test data.

Executing the complete instrumented program resulted in the path coverage information listed in Table 3.1. Path testing computer time (on the CDC 7600) for the complete test object was as follows (in seconds):

Instrumentation	30
Compilation of instrumented source	11
Loading object file	1
Execution using initial data	39
Coverage analysis	21
Total	102 seconds

```
----------------------------------------------------------------------------------------------------------------
MODULE  SPREDATAS              CUMULATIVE RESULTS OF     2 TEST CASES
----------------------------------------------------------------------------------------------------------------
DD PATH :  NO.  NOT EXECUTED  I   NUMBER OF EXECUTIONS -- NORMALIZED TO MAXIMUM              I       I NUMBER OF
NUMBER  :                     I .-------20.-------40.-------60.-------80.------100.          I       I EXECUTIONS
----------------------------------------------------------------------------------------------------------------
        I                     I
    1   I                     I                                                              I   1   I        1
    2   I                     I                                                              I   2   I        2
    3   I                     I                                                              I   3   I        1
    4   I                     I                                                              I   4   I        1
    5   I   5          00000  I                                                              I       I
    6   I   6          00000  I                                                              I       I
    7   I                     I                                                              I   7   I        1
    8   I                     I                                                              I   8   I        1
    9   I   9          00000  I                                                              I       I
  ...   I  ...         00000  I ... ... ...                                                  I       I
   11   I  11          00000  I                                                              I       I
   12   I                     I                                                              I  12   I        1
   13   I  13          00000  I                                                              I       I
  ...   I  ...         00000  I ... ... ...                                                  I       I
   69   I  69          00000  I                                                              I       I
   70   I                     I                                                              I  70   I        3
   71   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  71   I       87
   72   I  72          00000  I                                                              I       I
   73   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  73   I       87
   74   I  74          00000  I                                                              I       I
   75   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  75   I       87
   76   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  76   I       87
   77   I  77          00000  I                                                              I       I
  ...   I  ...         00000  I ... ... ...                                                  I       I
   81   I  81          00000  I                                                              I       I
   82   I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                   I  82   I      145
   83   I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                            I  83   I      116
   84   I                     I XXXXXXXX                                                     I  84   I       29
   85   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  85   I       87
   86   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  86   I       87
   87   I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX           I  87   I      174
   88   I  88          00000  I                                                              I       I
   89   I                     I XXXXXXXXXXXXXXXXXXXXXXXXX                                    I  89   I       87
   90   I                     I                                                              I  90   I        1
   91   I                     I XXXXXXXX                                                     I  91   I       28
   92   I                     I                                                              I  92   I        3
   93   I                     I XXXXXXX                                                      I  93   I       26
   94   I                     I                                                              I  94   I        1
   95   I                     I                                                              I  95   I        1
   96   I                     I                                                              I  96   I        1
   97   I  97          00000  I                                                              I       I
  ...   I  ...         00000  I ... ... ...                                                  I       I
   99   I  99          00000  I                                                              I       I
----------------------------------------------------------------------------------------------------------------

                                                        TOTAL NUMBER OF DD PATH EXECUTIONS =      1145
----------------------------------------------------------------------------------------------------------------
  TOTAL OF 73  NOT EXECUTED      EXECUTED  26/ 99         PERCENT EXECUTED =     26.26
```

Figure 3.1. Path Coverage Report

TABLE 3.1

PATH COVERAGE OF TEST OBJECT USING INITIAL DATA SET

Module	Paths Hit	Total Paths	Percent Coverage	Module	Paths Hit	Total Paths	Percent Coverage
PRIMER4	5	5	100	ORBP	10	22	45
ADIV	5	5	100	ORBTIM	11	24	46
ALTF	2	4	50	ORBTIME	2	4	50
AZF	2	3	67	ORB1	2	3	67
CAXIAL	5	5	100	* ORB2	10	62	16
CEASE	1	1	100	✓ ORIO	15	47	32
CHEKFIL	3	6	50	OUTCOL	26	31	84
CNORML	1	1	100	OUTSET	19	21	90
COUNOUT	2	3	67	* PREDATA	26	99	26
CROSS	4	4	100	SEPA	5	7	71
DATEF	1	1	100	SETKORD	3	13	23
DISCON	10	12	84	SONIC	10	19	53
DNSITY	5	8	63	STALE	9	39	23
DOT	1	1	100	* STIN	23	47	49
DTIMEF	1	1	100	STINT	5	27	19
ELF	2	3	67	✓ STOUT	46	91	51
ENDDOC	1	1	100	STREP	14	17	83
EULANG	13	33	39	SUBHEAD	3	16	19
FLAC	54	94	57	MISTAKE	0	4	0
FLIER	15	23	65	OLDATA	0	1	0
* FLIGHT	43	63	68	Q8ERROR	0	13	0
* FLIN	44	89	49	SUBVEC	1	1	100
FOAL	26	53	49	TITLER	11	15	73
GRAV	6	8	75	TRNSFM	17	19	89
* HEAD	25	61	41	UNITV	3	3	100
INCOL	25	39	64	VECLIN	3	3	100
IN1	19	25	76	WRITIT	22	29	76
KONVERG	20	46	43	XMAG	1	1	100
LSKIP	7	7	100				
				Totals	645	1,283	50

3.2.2 Retesting Strategy

The task of increasing path coverage is easily subdivided on a per-module basis. Several of SQLAB's documentation reports provide additional information for managing the testing activity. For example, the wrap-up report, shown in Fig. 3.2, lists the number of statements and the number of paths in each module. The invocation bands reports show module dependencies and the calling structure of the program which are also helpful. These reports can be generated for each module in the system. One is shown in Fig. 3.3.

Choosing test targets for expanding a data set should be based on software function, location in the module hierarchy, path coverage derived from existing data, and other factors particular to the test object. A general path testing-based methodolody is given in the JAVS User's Guide.[1]

For this test object, eight of the largest (in terms of FORTRAN statements) and highest level (in terms of module control hierarchy) were selected as targets. These modules are the starred and checked modules in Table 3.1. Using the initial data set, most of the modules had fairly low path coverage. It was found by inspection that, due to the data passed to them by the main program, modules ORIO and STOUT would never achieve much higher path coverage unless they were removed from the test object environment and driven separately. Thus they were omitted as test targets for the purpose of expanding the data set.

Path coverage of the remaining six modules was used as a basis for data set expansion. Several additional data sets were derived, and the resulting path coverage is shown in Table 3.2.

---

[1] C. Gannon and N. B. Brooks, JAVS Technical Report, Vol. 1: User's Guide, General Research Corporation CR-1-722/1, June 1978

NO.	NAME	TYPE	MODE	LANGUAGE	STMTS	ARGS	ENTRS	CMNKS	EQUIV	READS	WRITS	DLPS
1	PRIMER4	PROG	TYPELESS	FORTRAN	57	6	1	0	0	0	0	5
2	ADIV	FUNC	REAL	FORTRAN	7	2	1	0	0	0	0	5
3	ALTF	FUNC	REAL	FORTRAN	13	1	1	2	0	0	0	4
4	AZF	FUNC	REAL	FORTRAN	8	1	1	0	0	0	0	3
5	CAXIAL	FUNC	REAL	FORTRAN	18	5	1	0	0	0	0	5
6	CEASE	SUBR	TYPELESS	FORTRAN	5	1	1	0	0	0	0	1
7	CHEKFIL	FUNC	REAL	FORTRAN	20	1	4	1	1	0	0	6
8	CNORML	FUNC	REAL	FORTRAN	5	5	1	0	0	0	0	1
9	COUNOUT	SUBR	TYPELESS	FORTRAN	7	1	1	1	0	0	0	3
10	CROSS	SUBR	TYPELESS	FORTRAN	19	3	2	0	0	0	0	4
11	DATEF	FUNC	REAL	FORTRAN	5	1	1	0	0	0	0	1
12	DISCON	SUBR	TYPELESS	FORTRAN	20	6	1	0	0	0	0	12
13	DNSITY	FUNC	REAL	FORTRAN	22	1	2	2	0	0	0	8
14	DOT	FUNC	REAL	FORTRAN	6	2	1	0	0	0	0	1
15	DTIMEF	FUNC	REAL	FORTRAN	5	1	1	0	0	0	0	1
16	ELF	FUNC	REAL	FORTRAN	10	1	1	0	0	0	0	3
17	EULANG	SUBR	TYPELESS	FORTRAN	112	4	1	1	3	0	0	33
18	FLAC	SUBR	TYPELESS	FORTRAN	180	3	1	3	3	0	0	94
19	FLIER	SUBR	TYPELESS	FORTRAN	54	4	1	2	2	0	0	23
20	FLIGHT	SUBR	TYPELESS	FORTRAN	164	11	1	2	2	0	2	63
21	FLIN	SUBR	TYPELESS	FORTRAN	191	4	1	3	1	2	0	89
22	FOAL	FUNC	REAL	FORTRAN	121	2	1	2	0	0	1	53
23	GRAV	SUBR	TYPELESS	FORTRAN	19	2	1	2	0	0	0	8
24	HEAD	SUBR	TYPELESS	FORTRAN	165	1	1	3	1	2	16	61
25	INCCL	SUBR	TYPELESS	FORTRAN	87	5	1	1	1	2	0	37
26	IN1	SUBR	TYPELESS	FORTRAN	69	2	1	2	2	1	3	25
27	KONVERG	FUNC	INTEGER	FORTRAN	81	3	3	0	4	0	1	46
28	LSKIP	SUBR	TYPELESS	FORTRAN	21	1	1	1	0	0	1	7
29	MISTAKE	FUNC	INTEGER	FORTRAN	20	1	4	0	0	0	0	6
30	OLDATA	SUBR	TYPELESS	FORTRAN	7	1	1	1	0	0	1	1
31	ORBF	SUBR	TYPELESS	FORTRAN	77	4	1	1	2	0	1	22
32	ORBTIM	FUNC	REAL	FORTRAN	52	5	1	1	0	0	0	24
33	ORBTIME	FUNC	REAL	FORTRAN	12	3	1	1	0	0	0	4
34	ORB1	SUBR	TYPELESS	FORTRAN	26	2	1	1	2	0	0	3
35	ORB2	SUBR	TYPELESS	FORTRAN	182	5	1	1	1	0	1	62
36	ORIO	SUBR	TYPELESS	FORTRAN	119	5	1	3	2	2	1	43
37	OUTCOL	SUBR	TYPELESS	FORTRAN	62	5	1	2	0	0	4	27
38	OUTSET	SUBR	TYPELESS	FORTRAN	47	5	1	1	0	0	0	21
39	PREDATA	SUBR	TYPELESS	FORTRAN	249	1	2	1	0	5	18	97
40	SEPA	FUNC	REAL	FORTRAN	20	2	1	0	0	0	0	7
41	SETKORD	SUBR	TYPELESS	FORTRAN	44	1	5	2	1	0	4	13
42	SONIC	FUNC	REAL	FORTRAN	40	1	1	2	0	0	0	19
43	STALE	SUBR	TYPELESS	FORTRAN	82	4	1	3	3	0	0	39
44	STIN	SUBR	TYPELESS	FORTRAN	97	5	1	3	0	2	0	47
45	STINT	SUBR	TYPELESS	FORTRAN	55	0	1	3	3	0	0	19
46	STOUT	SUBR	TYPELESS	FORTRAN	159	5	1	3	4	0	0	91
47	STREP	SUBR	TYPELESS	FORTRAN	55	4	1	0	1	0	0	17
48	SUBHEAD	SUBR	TYPELESS	FORTRAN	34	2	1	1	0	0	1	14
49	SUBVEC	SUBR	TYPELESS	FORTRAN	7	3	1	0	0	0	0	1
50	TITLER	SUBR	TYPELESS	FORTRAN	44	2	1	1	0	0	1	15
51	TRNSFR	SUBR	TYPELESS	FORTRAN	39	5	1	0	0	0	0	13
52	UNITV	SUBR	TYPELESS	FORTRAN	8	2	1	0	0	0	0	3
53	VECLIN	SUBR	TYPELESS	FORTRAN	7	5	1	0	0	0	0	3
54	WRITIT	SUBR	TYPELESS	FORTRAN	67	3	1	0	1	0	1	29
55	XMAG	FUNC	REAL	FORTRAN	5	1	1	0	0	0	0	1
56	ENDDOC	FUNC	LOGICAL	FORTRAN	4	1	1	0	0	0	0	1
57	OBERROR	SUBR	TYPELESS	FORTRAN	29	2	1	0	0	0	2	9

Figure 3.2. SQLAB Wrap-up Report

```
SUBROUTINE STOUT(TITLE,KFORM,NAMES,STATES,LINES)

  -3        -2        -1         0         1         2         3
------------------------------------------------------------------
                               STOUT
                     FLIGHT              ABS
          PRIMER4                        IABS
                     FOAL                MINO
          FLIER                          MOD
          PRIMER4                        OUTCOL
                     ORB2                          HEAD
          PRIMER4                                  IABS
                     PRIMER4                       LSKIP
                     STIN                          TITLER
          PRIMER4                        OUTSET
                                                   MOD
                                                   XMIT
                                         STALE
                                                   AZF
                                                   COS
                                                   CROSS
                                                   DOT
                                                   ELF
                                                   MOD
                                                   SIN
                                                   UNITV
                                                   XMAG
                                                   XMIT
                                         STREP
                                                   ADIV
                                                   AZF
                                                   COS
                                                   ELF
                                                   IABS
                                                   SIN
                                                   SQRT
                                                   XMIT
                                         XMIT
------------------------------------------------------------------
```

Figure 3.8. SQLAB Invocation Bands Report

TABLE 3.2

PATH COVERAGE OF SELECTED MODULES

USING EXPANDED DATA SET

Module	Expanded Data Paths Hit	Total Paths	Initial Data % Coverage	Expanded Data % Coverage
FLIGHT	22	27	68	81
FLIN	50	89	49	56
HEAD	53	61	41	87
ORB2	38	62	16	61
PREDATA	94	99	26	95
STIN	31	47	49	66
	288	385	44%	75%

The first module for which new data was created was the data manipulation routine PREDATA. The coverage for this module was increased from 26 percent to 95 percent by adding two additional test cases to the original data set. This module is the largest TRAID routine (250 statements, 99 paths), and it was clear that it had not been very thoroughly exercised by the original data set. The results were less dramatic for other modules. The coverage for the routine that controls the guided missile flight was increased only 3.2 percent, from 68.2 to 71.4 percent. Coverage of subordinate modules, however, was significantly increased.

Finally, it should be noted that a number of program segments could not be reached by changing the input data. Many of the TRAID routines are general in purpose but are only used in a specific mode or for a specific feature. For example, 10 of the 63 paths in the flight control routine were found to be unreachable because of the main program construction. Other paths which lead to abnormal program termination were checked manually and are intentionally avoided during instrumented test runs. Path coverage results must, therefore, be interpreted carefully.

4 ERROR SEEDING

In generating errors in the test software several considerations were found appropriate:

1. To be realistic, the errors should be representative of those found in large programs in both type and frequency of occurrence.

2. The error types must be applicable to the test software and the test environment.

3. To evaluate test tools which utilize program execution, one or more errors should lead to abnormal program behavior for at least some test data.

The following subsections describe how error types were selected and their frequency determined, demonstrate how these criteria were applied to the test software in generating errors, and present the results of executing the software with seeded errors.

4.1 ERROR TYPES AND FREQUENCY

Several studies[1-3] have reported on the kinds and numbers of errors found in real-time programs. Of these, the data in the TRW study are directly applicable to the error-seeding experiment. We have used the Project 5 data from that work as the basis for the error types and their corresponding frequencies of occurrence.

---

[1] T. A. Thayer et.al, Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, Redondo Beach, California, August 1976.

[2] N. J. Fries, Software Error Data Acquisition, Boeing Aerospace Company, RADC-TR-77-130, Seattle, Washington, April 1977.

[3] Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974.

(1) There are several factors which limited the types of errors which were used for the experiment. The experiment was conducted on the existing software whose system requirements are not documented. (2) In that there is no time-critical or interactive requirement, the test software itself lacks certain characteristics of real-time programs. Rather the test environment has the test software executing as a normal batch job. (3) During path testing, certain test tool software is executed in conjunction with the test object software with added overhead. (4) The purpose of the experiment is to evaluate the use of test tools in locating errors in programs (not errors in specifications or documentation). Therefore, error types related to requirements, real-time performance, interactive usage, operating system interface, and software developmental procedures were not considered relevant to the experiment.

The project 5 data is based on a list of 79 error types shown in Table 4.1 grouped into twelve categories. In the TRW study only categories A through H and J resulted in code changes to the software. For the experiment, category J and error types D500, D700, D800, F400, F500, and F600 are not applicable to the test software and the test environment.

The first three columns of Table 4.2 contain error frequency data from Project 5. Listed for each major category (categories C and E were combined) are the number of errors resulting in code changes and the percent of total errors. Since category J is not applicable to the experiment, the percentages have been adjusted to those listed in column 5. In generating errors for the experiment, the applicable percentages were used as a goal for each major category. Column 6 lists the number of errors actually generated for the experiment and column 7 lists the number of errors which exhibited abnormal program behavior in the output from the test software when a single error was present.

Table 4.1. Error Types Used in Experiment

PROJECT 5 ERROR CATEGORIES*		Applicable to Experiment
A_000	COMPUTATIONAL ERRORS	✓
A_100	Incorrect operand in equation	✓
A_200	Incorrect use of parenthesis	✓
A_300	Sign convention error	✓
A_400	Units or data conversion error	✓
A_500	Computation produces over/under flow	✓
A_600	Incorrect/inaccurate equation used/wrong sequence	✓
A_700	Precision loss due to mixed mode	✓
A_800	Missing computation	✓
A_900	Rounding or truncation error	✓
B_000	LOGIC ERRORS	✓
B_100	Incorrect operand in logical expression	✓
B_200	Logic activities out of sequence	✓
B_300	Wrong variable being checked	✓
B_400	Missing logic or condition tests	✓
B_500	Too many/few statements in loop	✓
B_600	Loop iterated incorrect number of times (including endless loop)	✓
B_700	Duplicate logic	✓
C_000	DATA INPUT ERRORS	✓
C_100	Invalid input read from correct data file	✓
C_200	Input read from incorrect data file	✓
C_300	Incorrect input format	✓
C_400	Incorrect format statement referenced	✓
C-500	End of file encountered prematurely	✓
C_600	End of file missing	✓
D_000	DATA HANDLING ERRORS	✓
D_050	Data file not rewound before reading	✓
D_100	Data initialization not done	✓
D_200	Data initialization done improperly	✓
D_300	Variable used as a flag or index not set properly	✓
D_400	Variable referred to by the wrong name	✓
D_500	Bit manipulation done incorrectly	
D_600	Incorrect variable type	✓
D_700	Data packing/unpacking error	
D_800	Sort error	
D_900	Subscripting error	✓

*From Table 3-2 of TRW Study

Table 4.1. (Cont'd)

	PROJECT 5 ERROR CATEGORIES*	Applicable to Experiment
E_000	DATA OUTPUT ERRORS	✓
E_100	Data written on wrong file	✓
E_200	Data written according to the wrong format statement	✓
E_300	Data written in wrong format	✓
E_400	Data written with wrong carriage control	✓
E_500	Incomplete or msising output	✓
E_600	Output field size too small	✓
E_700	Line count or page eject problem	✓
E_800	Output garbled or misleading	
F_000	INTERFACE ERRORS	✓
F_100	Wrong subroutine called	✓
F_200	Call to subroutine not made or made in wrong place	✓
F_300	Subroutine arguments not consistent in type, units, order, etc.	✓
F_400	Subroutine called is nonexistent	
F_500	Software/data base interface error	
F_600	Software user interface error	
F_700	Software/software interface error	✓
G_000	DATA DEFINITION ERRORS	✓
G_100	Data not properly defined/dimensioned	✓
G_200	Data referenced out of bounds	✓
G_300	Data being referenced at incorrect location	✓
G_400	Data pointers not incremented properly	✓
H_000	DATA BASE ERRORS	✓
H_100	Data not initialized in data base	✓
H_200	Data initialized to incorrect value	✓
H_300	Data units are incorrect	✓
I_000	OPERATION ERRORS	
I_100	Operating system error (vendor supplied)	
I_200	Hardware error	
I_300	Operator error	
I_400	Test execution error	
I_500	User misunderstanding/error	
I_600	Configuration control error	

*From Table 3-2 of TRW Study

Table 4.1. (Cont'd)

PROJECT 5 ERROR CATEGORIES*		Applicable to Experiment
J_000	OTHER	
J_100	Time limit exceeded	
J_200	Core storage limit exceeded	
J_300	Output line limit exceeded	
J_400	Compilation error	
J_500	Code or design inefficient/not necessary	
J_600	User/programmer requested enhancement	
J_700	Design nonresponsive to requirements	
J_800	Code delivery or redelivery	
J_900	Software not compatible with project standards	
K_000	DOCUMENTATION ERRORS	
K_100	User manual	
K_200	Interface specification	
K_300	Design specification	
K_400	Requirements specification	
K_500	Test documentation	
X0000	PROBLEM REPORT REJECTION	
X0001	No problem	
X0002	Void/withdrawn	
X0003	Out of scope - not part of approved design	
X0004	Duplicates another problem report	
X0005	Deferred	

*From Table 3-2 of TRW Study

Table 4.2. Error Frequency in Major Categories

(1) Project 5 Major Error Categories*		(2) Total Project 5 Errors*	(3) Percent of Errors*	(4) Applicable to Experiment	(5) Percent Applicable	(6) Errors Generated	(7) Errors Manifested in Output
Computational	(A)	92	12.1	yes	14	14	7
Logic	(B)	169	24.5	yes	28	25	13
Data Input and Data Output	(C) and (E)	55	7.8	yes	9	9	6
Data Handling	(D)	65	11.0	yes	13	10	7
Interface	(F)	48	7.0	yes	8	7	4
Data Definition	(G)	62	8.9	yes	10	8	4
Data Base	(H)	112	16.2	yes	18	13	8
Other	(J)	86	12.5	no	--	--	--
		689	100.0		100	86	49

*Data derived from Table 4-2 of TRW study.

4.2 ERROR GENERATION

In addition to generating errors whose type and frequency have their bases in a published study, the location of each error and the program's resulting behavior were also prime concerns in maintaining an objective experiment. In the TRW study, no data linking the error type to software property (e.g., statement type) is presented. Using the error type made it necessary to establish correlations between each error type and quantifiable test software properties. Furthermore, since the test object consists primarily of general utility subroutines, many having alternative segments of code whose execution depends upon their input parameter data, we felt that the errors should reside on segments of code that are executed by a thorough (in terms of program function and structure) set of test data, and that the errors should manifest themselves by some deviation in the program's normal output. To generate errors with these properties, the following steps were performed.

1. The test software was analyzed by the test tool[1] to classify source statements, to obtain software documentation reference material (e.g., symbol set/usage, module interaction hierarchy, location of all invocations), to guide insertion of errors, and to generate an expanded set of test data that provided thorough path coverage. The percentage of path coverage varied from module to module depending upon the main program's application of the utility subroutines.

2. A matrix showing error types versus statement classification was manually derived.

3. The information from steps 1 and 2 was combined into a matrix showing potential sites in the software for each error type.

---

[1] D. M. Andrews and J. P. Benson, Software Quality Laboratory User's Manual, General Research Corporation CR-4-770, May 1978.

4. From the potential site matrix, a list of candidate error sites was randomly generated.

5. At each site in the list either an error of the designated type was manually inserted or the site was rejected as being unsuitable for the error type.

6. Errors were eliminated from the error set which caused a compiler or loader diagnostic.

7. The 86 errors shown in column 6 of Table 4.2 were selected from the remaining errors using Project 5 error frequency data. Errors from this set were eliminated if they caused no change in the output. Fifteen errors were rejected due to lack of coverage with the test data, and 22 were eliminated for which coverage was achieved without affecting the output. The surviving 49 errors, shown in column 7, were used.

Error site execution or reference was verified by an output message placed, for the case of executable statements, at the error site or, for the case of non-executable statements, at the site of reference by some executable statement on a covered path. The impact of this evidence is that path testing with the sole goal of execution coverage is <u>not</u> an adequate verification measure. Most software tool developers whose verification tools include a path testing capability advocate their usage with data that demonstrate all specific functions of the software. Even then, stress and other performance testing should enter into the total test plan.

4.2.1 Error Seeding Preliminary Analysis

Using the SQLAB tools, the original test software was processed to generate standard documentation and static analysis reports. The reports include the following:

1. A list of the software properties of each module with a count of each statement type and the characteristics of the interface
2. A listing of the source for each module in the test software
3. Source for all invocations to and from each module
4. Local and global cross reference lists indicating usage for all names
5. Path identification for each DD-path in each module
6. Hierarchy relationships between modules
7. Static checks on variable usage.

A master list of test software properties was constructed from item 1 and the linkage established between each software property and the error types. The linkages together with the data for each module were used to select candidate error sites. The other reports were used to generate actual errors. The following subsections explain how this was accomplished.

4.2.2 Software Property and Error Type Linkage

The master list of software properties constructed from item 1 (see Table 4.3) reflects the dialect of FORTRAN used (e.g., DECODE and ENCODE) and the statement types used in the test software (e.g., no DOUBLE PRECISION or PUNCH statements). Additionally, the list includes only those statement types relevant to the experiment (e.g., comment statements and END statements are omitted). Two interface properties are included, Parameter and Invocation, although there is some overlap with other constructs. The linkage between software properties and error types was established by listing, for each error type, all software properties that could be the site of an error of that type. These linkages are shown in Table 4.3.

TABLE 4.3

RELATIONSHIPS BETWEEN SOFTWARE PROPERTIES AND ERROR TYPES

Major Error Categories and Error Types

Test Software Property	A Computational									B Logic							C Input						E Output							D Data Handling							F Interface				G Data Definition				H Data Base		
Statement	A100	A200	A300	A400	A500	A600	A700	A800	A900	B100	B200	B300	B400	B500	B600	B700	C100	C200	C300	C400	C500	C600	E100	E200	E300	E400	E500	E600	E700	D050	D100	D200	D300	D400	D600	D900	F100	F200	F300	F700	G100	G200	G300	G400	H100	H200	H300
Assignment	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓			✓			✓	✓	✓			✓	✓	✓				✓				✓	✓	✓	✓	✓	✓	✓			✓	✓	✓		✓	
ASSIGN											✓																							✓	✓												
BACKSPACE																																		✓	✓												
CALL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								✓					✓	✓									✓	✓	✓	✓	✓	✓	✓		✓	✓	✓				
COMMON																																		✓	✓	✓					✓	✓	✓				
Computed GOTO											✓	✓	✓																					✓	✓												
CONTINUE											✓																																				
DATA															✓																✓	✓	✓	✓	✓	✓					✓	✓		✓	✓	✓	✓
DECODE					✓		✓		✓			✓			✓										✓								✓	✓	✓	✓						✓	✓	✓	✓	✓	✓
DIMENSION																																		✓	✓	✓					✓		✓				
DO											✓	✓	✓	✓	✓	✓																	✓	✓	✓	✓								✓			
ENCODE					✓		✓		✓			✓			✓										✓								✓	✓	✓	✓						✓	✓	✓	✓	✓	✓
ENTRY											✓																										✓		✓								
EQUIVALENCE																																		✓	✓	✓					✓		✓				
EXTERNAL																																		✓	✓		✓		✓	✓	✓						
FORMAT							✓		✓										✓						✓	✓	✓	✓	✓																		
FUNCTION																																		✓	✓		✓			✓							
Assigned GOTO											✓	✓	✓			✓																		✓	✓												
GOTO											✓		✓																																		
IF end-of-file											✓	✓	✓			✓																	✓	✓	✓	✓	✓		✓								
Three-branch IF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓																	✓	✓	✓	✓	✓	✓	✓			✓	✓				
Two-branch IF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓																	✓	✓	✓	✓	✓	✓	✓			✓	✓				
Logical IF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓																	✓	✓	✓	✓	✓	✓	✓			✓	✓				
INTEGER																																		✓	✓						✓						
LOGICAL																																		✓	✓						✓						
PRINT					✓		✓		✓			✓			✓								✓						✓				✓	✓	✓	✓						✓	✓	✓			
PROGRAM																			✓				✓											✓	✓		✓										
READ					✓		✓		✓			✓			✓		✓	✓		✓	✓	✓								✓			✓	✓	✓	✓				✓		✓	✓	✓			✓
REAL																																		✓	✓						✓						
RETURN											✓																													✓							
REWIND																														✓				✓	✓												
STOP											✓		✓																											✓							
SUBROUTINE																																		✓	✓		✓			✓							
WRITE					✓		✓		✓			✓			✓								✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	✓				✓		✓	✓	✓			✓
Interface																																															
PARAMETER	✓									✓		✓											✓	✓										✓	✓	✓	✓		✓	✓			✓				✓
INVOCATION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓													✓	✓									✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓

The linkages were manually generated. In some instances, syntactic and semantic rules for FORTRAN were used to determine entries. For example, any statement type in which an arithmetic expression is permitted (e.g., Assignment, CALL, IF) is a possible site for an error in the computational category (error types A100 through A900). Similarly, FORMAT, READ, PRINT, WRITE, DECODE, and ENCODE statements are possible sites for input and output error types in categories C and E involving data conversion.

Other entries in the table indicate statement types which are directly associated with an error type, although an error may involve a combination or sequence of statements including other types not marked. For example, error type B500 (too many/few statements in loop) is directly associated with a DO statement (marked in the table) combined with at least one of any other executable statement (not marked).

In some instances, entries reflect how the test software processing is accomplished, although it may not be a common usage of the language. An example of this is the usage of assignment statements to construct variable formats, thereby linking the Assignment Statement type to error types C300 (incorrect input format) and E300 (Data written in wrong format).

4.2.3 Candidate Error Site Selection

The method used for error selection attempts to be realistic by utilizing published error types and frequencies (Table 4.2) while remaining objective by selecting placement by random. The test software contains over 50 modules. For each module, data showing the count of each software property was collected from SQLAB reports (see Sec. 4.2.1) into a matrix of the form shown in Fig. 4.1: This matrix, when combined with the matrix linking software property to error type (Table 4.3) yields a matrix of candidate error sites for each error type in each module. The form of the candidate error site matrix as shown in Fig. 4.2 is sub-divided according to major error category.

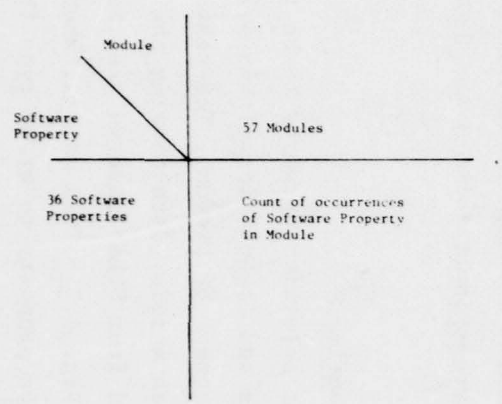


Figure 4.1. Form of Software Property/ Module Matrix

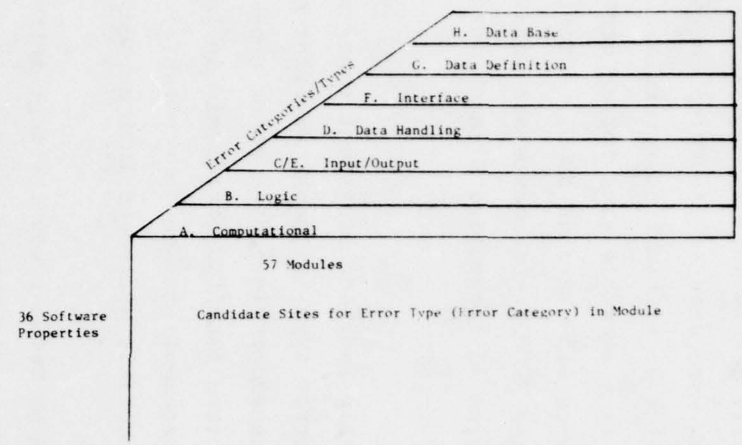


Figure 4.2. Candidate Error Site Matrix

For each major error category a randomly selected list of candidate error sites was generated using a simple computer program to perform the necessary computations for site selection. Input to the program consisted of the following data:

1. Error Category and Error Type List (Table 4.1).
2. A list of Software Property Names (See Table 4.3).
3. A list of Module Names (from SQLAB reports).
4. Error Category Frequency (Col. 5 of Table 4.2).
5. Software Property and Error Type Linkages (Table 4.3).
6. Software Property and Module Matrix (from SQLAB reports).
7. The number of error sites to generate.
8. Possible causes for each error type (statement sequence omitted or extra statement, input data)

The site selection program contains no algorithms to reject a selected site which is unsuitable for a particular error type (e.g., an assignment statement without any parenthesis for error type A200, Incorrect use of parenthesis). To provide for manual rejection of unsuitable sites, the number of sites was chosen to be twenty times the targeted number (50) for the experiment, or 1000 sites.

Output from the program consists of a list of the randomly selected candidate error sites for each major category. The number of sites generated for each category is proportional to the error frequency for the category, with the total number of sites equal to the desired number. The output for each candidate site identifies the site by module name, software property, the property's sequence number within the module, and the error type with its description. In addition, the possible causes for the error type are listed. Fig. 4.3 contains an excerpt reproduced from the output for major error category A, Computational. How this list was used to generate errors is explained in the following subsection.

```
CANDIDATE SITES FOR A000 COMPUTATIONAL
SELECT 14 SITES
MODULE      SITE        NUMBER      ERROR      DESCRIPTION
STREP       ASSIGNMENT  7           A100       INCORRECT OPERAND IN EQUATION
...POSSIBLE CAUSE IS STATEMENT SEQUENCE
PREDATA     IF          16          A800       MISSING COMPUTATION/WRONG SEQUENCE
...POSSIBLE CAUSE IS STATEMENT SEQUENCE
...POSSIBLE CAUSE IS OMITTED OR EXTRA STATEMENT
IN1         ASSIGNMENT  15          A900       ROUNDING OR TRUNCATION ERROR
...POSSIBLE CAUSE IS STATEMENT SEQUENCE
FLIGHT      CALL        6           A300       SIGN CONVENTION ERROR
STIN        IF          1           A200       INCORRECT USE OF PARENTHESIS
ORBP        IF-TWO      1           A300       SIGN CONVENTION ERROR
```

Figure 4.3. Excerpt From Candidate Error Site List

4.2.4 Error Set Generation

The task of generating a representative set of errors for the experiment consisted of three major steps:

Step 1. Using the candidate error site list as a guideline, a set of error packets was created which contained a selection of errors in the desired frequency for each of the major error categories.

Step 2. Error packets resulting in compiler or loader error messages or warnings were eliminated from the set.

Step 3. The acceptable error packets were applied, one at a time, to the source program and the results of executing the erroneous program analyzed and classified for later use in the experiment.

These three steps were repeated one time to obtain the final set of error packets used in the experiment.

Error Packets

Step 1 in this process was performed by repeating for each major error category the following sequence until the desired number of errors were generated:

1. Choose the next (initially, the first) site in the candidate site list (See Fig. 4.3).
2. Locate the site in the source program listing (e.g., the seventh assignment statement in STREP). Reject site if previously accepted; otherwise, continue.
3. Determine if error type is applicable to site (e.g., Would a change in operand be a likely error in the statement?). If not, reject site; otherwise, continue.

4. If site is an executable statement, determine whether statement was executed with test data using coverage reports from test software coverage analysis activity (Sec. 4.3.2.2). If site is a declaration statement, determine, if possible, whether information in declaration was referenced by using coverage reports. Accept site and continue if criteria met; otherwise, reject site.

5. Generate error packet for acceptable site and mark site to avoid duplication.

Each error packet includes the following items:

1. A unique, randomly selected packet identification name.

2. A code change constituting the error.

3. A print message identifying on the output the error by packet identification name (added as the first executable statement of the main program).

4. A comment statement identifying the error site and type (added at the error site).

5. A print message to record when the module containing the error is entered (added as the first executable statement of the module entry).

6. A print message to record when the error site is executed (added at the error site or at the location where the error is effective).

An example error packet is shown in Fig. 4.4. The system utility UPDATE was used to manage the error packets. Each item consists of one or more UPDATE directives (first character is *) and FORTRAN source text. The UPDATE directive serves to identify the packet (*ID) or to insert test (*I) or delete and insert text (*D) at a designated place in the

Item	Contents	Description	Used for Experiment	Used for Error Effect Analysis
1	*ID ERR007	Error packet identification	yes	yes
2	*D STREP.38 SP(3) = AZF(Y)	Change line 38 in STREP	yes	yes
3	*I PRIMERA.15 PRINT 2007 2007 FORMAT(* ERROR E007*)	Error packet identification message	yes	yes
4	*I STREP.38 C STREP ASSIGNMENT 7 A100	Error site and error type identification	no	yes
5	*I STREP.28 PRINT 30007 30007 FORMAT(*...MODULE ENTERED FOR E007*)	Module entry message	no	yes
6	*I STREP.38 PRINT 20007 20007 FORMAT(*...ERROR SITE EXECUTED FOR E007*)	Site execution message	no	yes

Figure 4.4. Sample Error Packet

test software. The set of error packets was placed in ascending order by the (randomly selected) packet names before input to the UPDATE utility.

The complete error packet was used to analyze the effect of the presence of each error prior to use in the experiment. For the experiment only the first three items in each packet were used to modify the software. One or more error packets were selected, then the source of the complete program including the errors was made available to the tester in a form which concealed the site and type of error (See Secs. 5 and 6.

Compiler and Loader Qualification

Step 2 in the error set generation process served to eliminate from the error set those errors which were revealed by the compiler or loader. The complete set of error packets was applied to the source program and the erroneous source compiled and executed. Error packets which resulted in compiler or loader error messages or warnings were eliminated from the set. A warning of an unset variable is an example of a compiler message which caused rejection of an error packet; similarly, an unsatisfied external warning by the loader caused rejection.

Error Analysis

Step 3 was to analyze the effect of the presence of each error during execution. The test software, with one error jacket applied, was compiled and executed with the sample test data obtained from preliminary coverage analysis. The output was then examined for print messages from items 5 and 6 of the error packet. In addition, comparisons were made to normal program output obtained by executing the error-free test software with the same data. The results of each error run were classified in one of the following categories:

Error	Code	Output Problem
	1 = no effect 2 = module executed 3 = site executed 4 = erroneous output	
E001	4	Orbit parameters are nonsense for ORB2 mode 1
E002	4	Trajectory printout is nonsense; miss distance okay
E003	4	Interceptor buries itself below surface
E004	3	-----
E005	2	-----
E006	3	-----
E007	4	Longitude and latitude printout the same
E021	4	Garbage on output
E027	4	Program loops printing I.D. page
E053	4	Program stops prematurely
E058	4	Program failed to reach solution

Figure 4.5. Selected Entries from Results List

1. No observed effect on normal output.
2. Module containing error executed with no observed effect on normal output.
3. Module containing error executed and error site executed with no observed effect on normal output.
4. Module containing error executed and error site executed with error manifested by differences in error run output from normal output.

Errors in category 4 were used in path testing portion of the experiment. Errors in all categories were used in other portions of the experiment.

For errors used in path testing, a "user complaint" about the erroneous output was prepared. The output problems included not only premature termination of program execution, but also discrepancies in user-expected program behavior, output format, and numeric results. Selected entries from a list of error packet names and results, prepared for use in the experiments, are shown in Fig. 4.5.

A total of 86 errors were generated; of these, 49 errors were manifested by erroneous output. A breakdown by error type is shown in Table 4.4. These are also summarized by major error category in Table 4.2 together with the error frequency data.

Table 4.5 shows the distribution of errors by software property and major error category; the total occurrences of the software property in the test software is also shown. Each non-blank entry represents a statement property linked to a major error category. Each non-zero entry is the count of error packets generated or manifested in the output.

Table 4.6 shows the distribution of errors by count of error packets in single module and cumulative error run results. Of 57 modules in the test software, 86 error packets were generated in 33 modules. Ten

TABLE 4.4

ERROR RUN RESULTS BY ERROR TYPE

Category	Error Packets Generated	Errors Manifested in Output
A. Computational		
A100	2	2
A200	3	2
A300	1	0
A400	1	1
A500	2	0
A600	2	0
A700		
A800	3	2
A900		
	14	7
B. Logic		
B100	2	1
B200	7	1
B300	3	3
B400	4	3
B500	5	4
B600	2	1
B700	2	0
	25	13
C/E. Input/Output		
C100		
C200	2	2
C300		
C400		
C500		
C600		
E100		
E200	1	1
E300	1	1
E400		
E500	1	1
E600	4	1
E700		
	9	6
D. Data Handling		
D050	1	1
D100	1	1
D200	4	2
D300		
D400	3	2
D600		
D900	1	1
	10	7
F. Interface		
F100		
F200	4	2
F300		
F700	3	2
	7	4
G. Data Definition		
G100	2	2
G200	2	2
G300	2	0
G400	2	0
	8	4
H. Data Base		
H100	3	2
H200	5	2
H300	5	4
	13	8
	86	49

TABLE 4.5

ERRORS CLASSIFIED

Software Property	Occurrences in Test Object	Errors Packets Generated for Major Error Category								Errors Manifested in Output							
Statement		A	B	C/E	D	F	G	H	Total	A	B	C/E	D	F	G	H	Total
Assignment	965	8	1	2	1	1	5	5	23	4	1	2	1	1	2	2	13
ASSIGN	28		1		0				1		1		0				1
BACKSPACE	1				0				0				0				0
CALL*	259	1	0	0	0	0	1	0	2	0	0	0	0	0	0	0	0
COMMON	56				0		0		0				0		0		0
Computed GOTO	12		0		0				0		0		0				0
CONTINUE	97		1						1		0						0
DATA	60		0		5		0	3	8		0		3		0	2	5
DECODE	6	0	0	0	0		0	1	1	0	0	0	0		0	1	1
DIMENSION	53				0		0		0				0		0		0
DO	91		5		1		0		6		4		1		0		5
ENCODE	4	0	0	0	0		0	0	0	0	0	0	0		0	0	0
ENTRY	14		1			0			1		0			0			0
EQUIVALENCE	39				0		0		0				0		0		0
EXTERNAL	1				0	0	0		0				0	0	0		0
FORMAT	55	0		4					4			2					2
FUNCTION	19				0	0			0				0	0			0
Assigned GOTO	17		0		0				0		0		0				0
GOTO	330		4						4		1						1
IF end-of-file	13		0		0	0			0		0		0	0			0
Three-branch IF	13	0	0		0	0	0		0		0		0	0	0		0
Two-branch IF	1	0	1		0	0	0		1		1		0	0	0		1
Logical IF	388	3	10		1	2	0		16	1	5		1	0	0		7
INTEGER	1				0		0		0				0		0		0
LOGICAL	11				0		0		0				0		0		0
PRINT	2	0	0	0	0		0		0	0	0	0	0		0		0
PROGRAM	1			0	0	0			0			0	0	0			0
ROAD	16	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1
REAL	1				0		0		0				0		0		0
RETURN	126		1			1			2		0			1			1
REWIND	11				1				1				1				1
STOP	1		0			0			0					0			0
SUBROUTINE	35				0	0			0				0	0			0
WRITE	55	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
Interface																	
Parameter	154	0	0	1	0	2	0	0	3	0	0	1	0	1	0	0	2
Invocation	548	2	0	0	1	1	2	4	10	2	0	0	0	1	2	3	8
	Total	14	25	9	10	7	8	13	86	7	13	6	7	4	4	8	49

TABLE 4.6

DISTRIBUTION OF ERRORS IN MODULES

Errors per Module	Error Packets Generated (Categories 1,2,3&4)		Erroneous Module Executed (Categories 2,3&4)		Error Site Executed (Categories 3&4)		Error Manifested in Output (Category 4)	
	Modules	Errors	Modules	Errors	Modules	Errors	Modules	Errors
1	10	10	8	8	10	10	11	11
2	12	24	12	24	9	18	9	18
3	3	9	3	9	4	12	2	6
4	4	16	4	16	2	8	1	4
5	0				1	5	2	10
6	2	12	2	12	3	18		
7	1	7	1	7				
8	1	8	1	8				
Total	33	86	31	84	29	71	25	49
modules %*	58		55		51		44	
errors %*		100		98		83		47

* 57 modules and 86 errors

modules had only one packet and no modules had more than eight. During single-error runs, modules containing 84 of the 86 errors were executed in 31 of the 33 error-seeded modules (two were contained in error-recovery routines not executed for the sample test data). The error site was executed for 71 of the 86 errors in 29 modules; but the error was manifested by the output in only 49 of the 86 error runs in 25 modules.

Note the large drop (22) in the number of errors manifested in output from the number whose site was executed (49 from 71). It is not uncommon for software containing errors to produce the "right" output even if the site of the error is executed. Upon analysis, these errors, although potentially dangerous, proved to be harmless in the test environment. For example, one caused calculations to be needlessly repeated, another preset data which was later reset before being used, and a third performed calculations whose results were never used. All three of these errors were time-consuming errors which could affect real-time responses. Table 4.7 lists the reasons these 22 errors resulted in acceptable output.

TABLE 4.7

CATEGORY 3 ERRORS (SITE EXECUTED)

Reason Error Not Manifested	Number of Errors
Variable value(s) acceptable	8
Variable reset before use on path taken	5
Loop executed only once	3
Statement sequence has no effect for path taken	2
Timing not critical	2
Variable not used after set	1
Input data complete	1
	22

5 SINGLE-ERROR EXPERIMENT

5.1 DESCRIPTION

Errors from the seven major categories were seeded, one at a time, into the FORTRAN program according to the frequencies shown in Table 5.1 and Fig. 5.1. For each error the analyst was given a compilation and execution listing which gave no clues to the error's location. He was told what was wrong with the output and had, as a specification of the proper program performance, a listing of the correct output. The task was to find the error using execution coverage analysis (path testing) or inspection, whichever seemed more appropriate, correct the source, and execute the corrected program to verify the output. Human and computer times were accounted for from the time the tester received the erroneous listing to the time he delivered the corrected listing.

To evaluate the types of errors detected by static analysis, all 49 errors were simultaneously seeded into the program after determining that they did not interfere with each other in the static sense. Only one computer run was required for this evaluation.

Unlike static analysis, which explicitly detects inconsistencies and locates the offending statement(s), path testing is a technique that demands skill to interpret the execution coverage data as well as to recognize improper program performance from the program's output.

5.2 PATH-TESTING PHASE

For the path-testing evaluation phase, we found that the errors were located using three detection methods: path testing alone, inspection aided by path testing, and inspection alone. Some errors were easily detected without the necessity of instrumenting the code to get path coverage. Some errors were found when the path coverage reports narrowed the search to a set of suspicious paths--but then

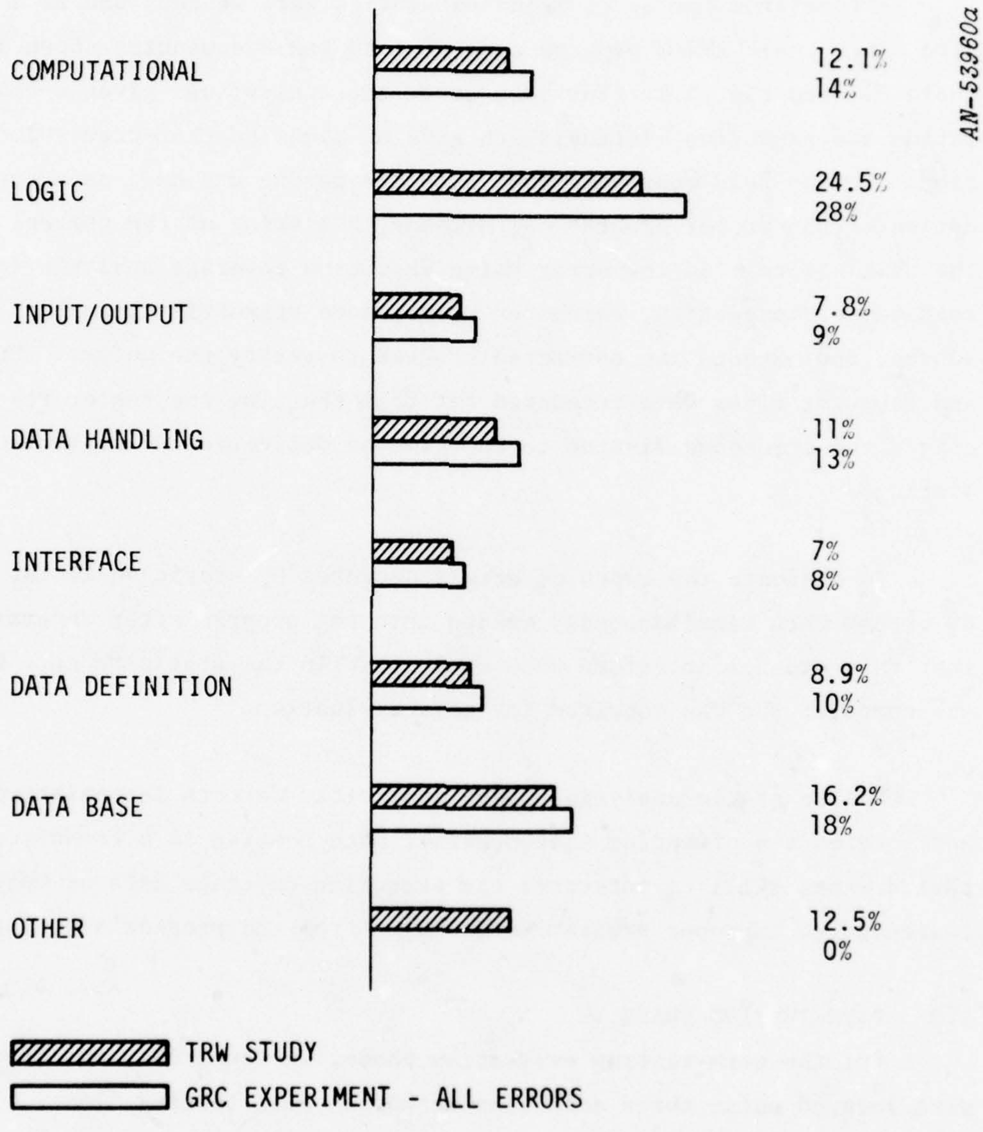


Figure 5.1. Error Frequency in Major Categories

TABLE 5.1

ERROR FREQUENCY IN MAJOR CATEGORIES

(1)	(2)	(3)	(4)	(5)	(6)
Project 5 Major Error Categories*	Total Project 5 Errors*	Percent of Errors*	Percent Applicable	Errors Generated	Errors Manifested in Output
Computational (A)	92	12.1	14	14	7
Logic (B)	169	24.5	28	25	13
Data Input (C) and Data Output (E)	55	7.8	9	9	6
Data Handling (D)	65	11.0	13	10	7
Interface (F)	48	7.0	8	7	4
Data Definition (G)	62	8.9	10	8	4
Data Base (H)	112	16.2	18	13	8
Other (J)	86	12.5	--	--	--
	689	100.0	100	86	49

*Data derived from Table 4-2 of TRW study.

inspection was used to actually determine the error. Other errors were found directly by observing the control path behavior from the coverage reports and the path statement definition listing. In a few cases the wrong "error" was found and only some of the incorrect symptoms disappeared (these are noted in Table 5.2).

Figure 5.2 shows the frequencies of error categories detected by the methods described above. The dashed lines show the effect of some degree of path testing coverage by reporting the sum of path testing alone and inspection aided by path testing. As one might expect, logic errors and computation errors (since they often cause a change in control flow) are the best candidates for path testing. Errors in these two categories are often the most difficult to locate, unless a detailed design and specification are also available. Input/output and data definition errors are usually easily detected by inspection alone.

More comprehensive results are shown in Table 5.2. Note that not all minor error types were seeded into the program, owing to project limitations. For each error seeded, Table 5.2 shows the technique used to detect it. An asterisk next to the technique's indicator signifies that the erroneous statement was located but the "correction" was not the proper one, or that more information (such as a specification) was needed to make the proper changes.

To assess the value of path testing, an account was kept of the resources expended. The average engineering time in hours for finding each error is shown in Fig. 5.3. Most of the errors detected by inspection required only about 1 1/2 hours to find and correct. On the other hand, the more difficult errors requiring path testing took about 4 hours.

TABLE 5.2

ERROR DETECTION FOR EACH ERROR TYPE

	Category	Static Analysis	Path Testing Phase	Errors Manifested in Output
A.	COMPUTATIONAL			
A100	Incorrect operand in equation		P,A	2
A200	Incorrect use of parenthesis	S	P,O	2
A400	Units or data conversion error		I	1
A300	Missing computation		A,O	2
		1	5	7
B.	LOGIC			
B100	Incorrect operand in logical expression		P*	1
B200	Logic activities out of sequence		P	1
B300	Wrong variable being checked		I,I,P	3
B400	Missing logic or condition tests	S,S	P,A,O	3
B500	Too many/few statements in loop		P,A,A,O	4
B600	Loop iterated incorrect number of times (including endless loop)		A	1
		2	10	13
C/E.	INPUT/OUTPUT			
C200	Input read from incorrect data file		I,I	2
E200	Data written according to the wrong format statement	S	P	1
E300	Data written in wrong format		I	1
E500	Incomplete or missing output		I	1
E600	Output field size too small		I	1
		1	6	6

	Category	Static Analysis	Path Testing Phase	Errors Manifested in Output
D.	DATA HANDLING			
D050	Data file not rewound before reading		I	1
D100	Data initialization not done	S	I	1
D200	Data initialization done improperly		I,I	2
D400	Variable referred to by the wrong name	S	A,O	2
D900	Subscripting error		P	1
		2	6	7
F.	INTERFACE			
F200	Call to subroutine not made or made in wrong place		P,I*	2
F700	Software/software interface error	S	I,I	2
		1	3	4
G.	DATA DEFINITION			
G100	Data not properly defined/dimensioned	S	I,I	2
G200	Data referenced out of bounds		A,I	2
		1	4	4
H.	DATA BASE			
H100	Data not initialized in data base		P,I	2
H200	Data initialized to incorrect value		A,I	2
H300	Data units are incorrect		P,I,A*,O	4
		0	6	8
	Total	8	40	49

S = Static Analysis (16% of total errors)

P = Path Testing only (25% of total errors)

A = Inspection aided by Path Testing (20% of total errors)

I = Inspection only (41% of total errors)

O = Error not detected (16% of total errors)

* = Error site located or improper correction made (counted as not detected)

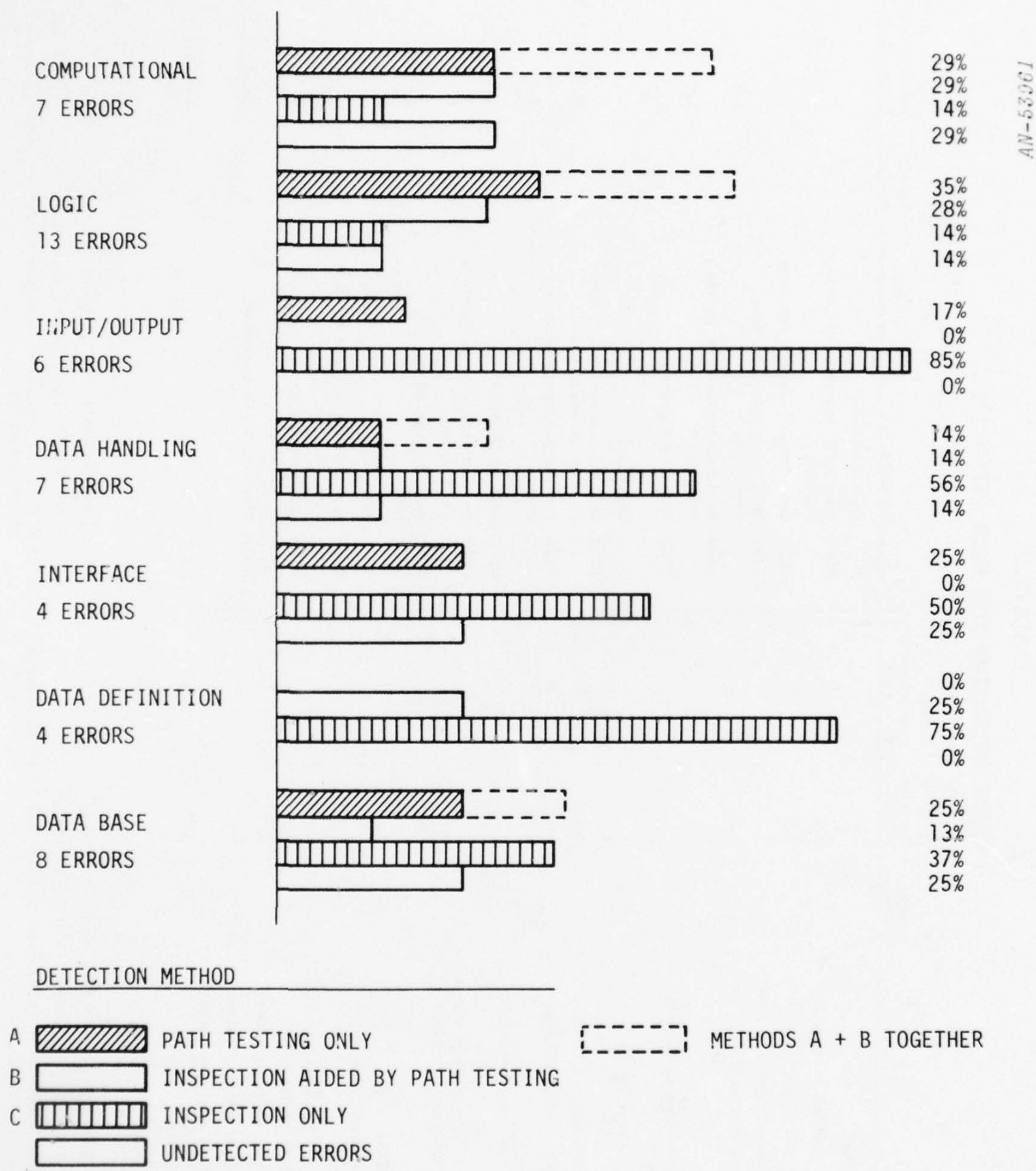
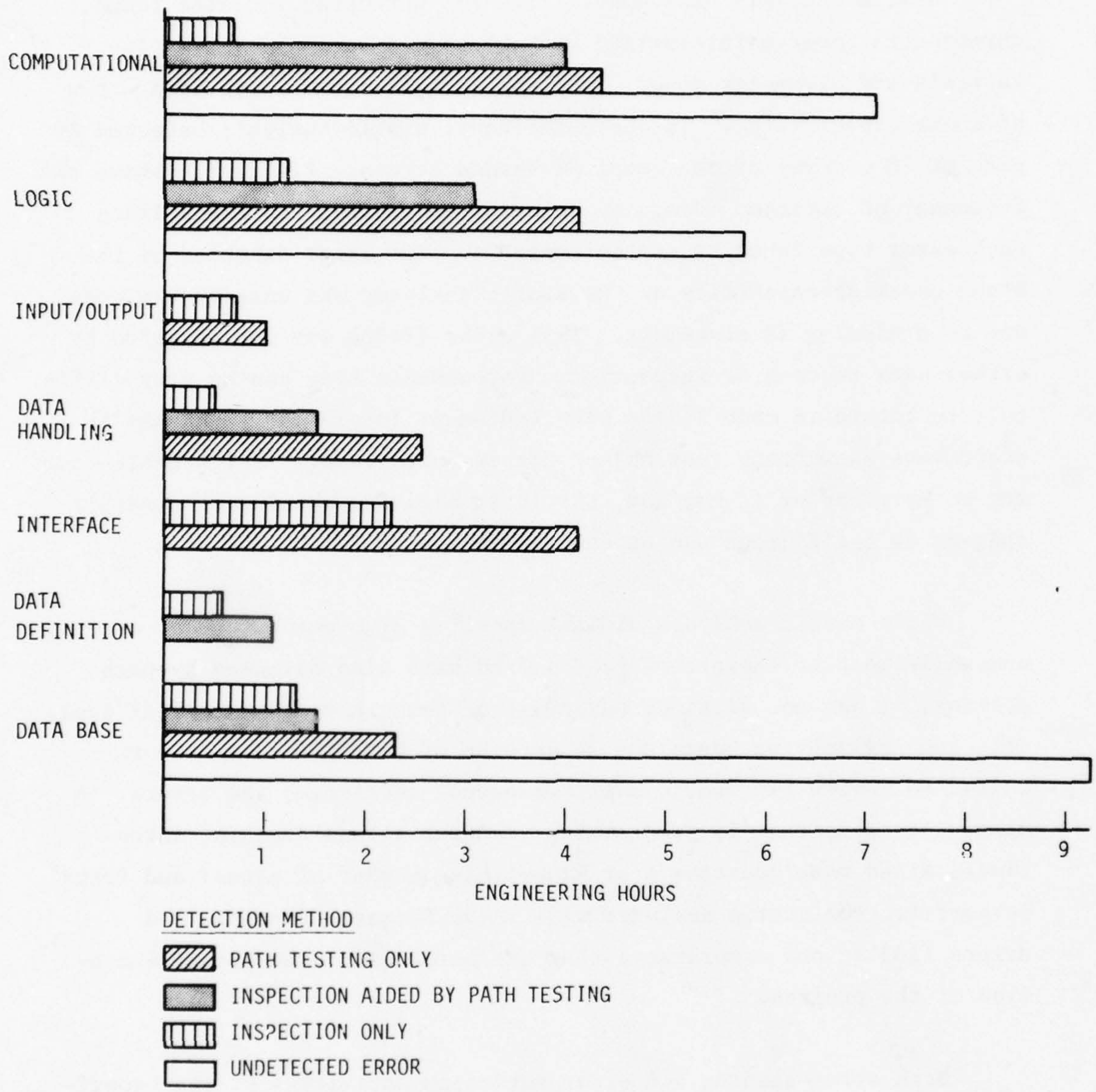


Figure 5.2. Path Testing Frequency of Detected Errors By Category



AN-63067

Figure 5.3. Path Testing: Average Time Expended per Error

5.3 STATIC ANALYSIS PHASE

Static analysis has capabilities for detecting infinite loops, unreachable code, uninitialized variables, and inconsistencies in variable and parameter mode. Some sophisticated compilers have a few of these capabilities. In our experiment, static analysis detected 16 percent (8 errors) of the total 49 seeded errors. Figure 5.4 shows the frequency of detected errors by major category, and Table 5.2 lists each error type found by static analysis. One error detected by the graph checking capability of the static analyzer was unreachable code due to a missing IF statement. This error (B400) was not detected by either path testing or inspection. Unreachable code can be very difficult to locate in code filled with statement labels and three-way IF statements as was the test object for the experiment. Unreachable code may be harmless or it may not, but it is always a warning of possible dangers or inefficient use of computer resources.

While static analysis did not detect a high percentage of errors, and while most of the errors it did find were also detected by path testing, it has the distinct advantage of being a very economical tool. Only two engineering hours and 24 seconds of CDC 7600 time were required to review the static analysis output and locate the errors. A disadvantage is that if programming practice allows frequent intentional mixed mode constructs or mismatching number of actual and formal parameters, the static analyzer will issue frequent warnings and errors (133 in our experiment) that are harmless to the proper execution of the program.

Both error-seeding and error detection activities of the experiment provided concrete data for several conclusions about the two testing techniques. While the experiment was designed and implemented in an objective manner and can be repeated by other interested researchers, it is not our intention to apply a metric or statistical significance to the error detection capabilities of the testing methods.

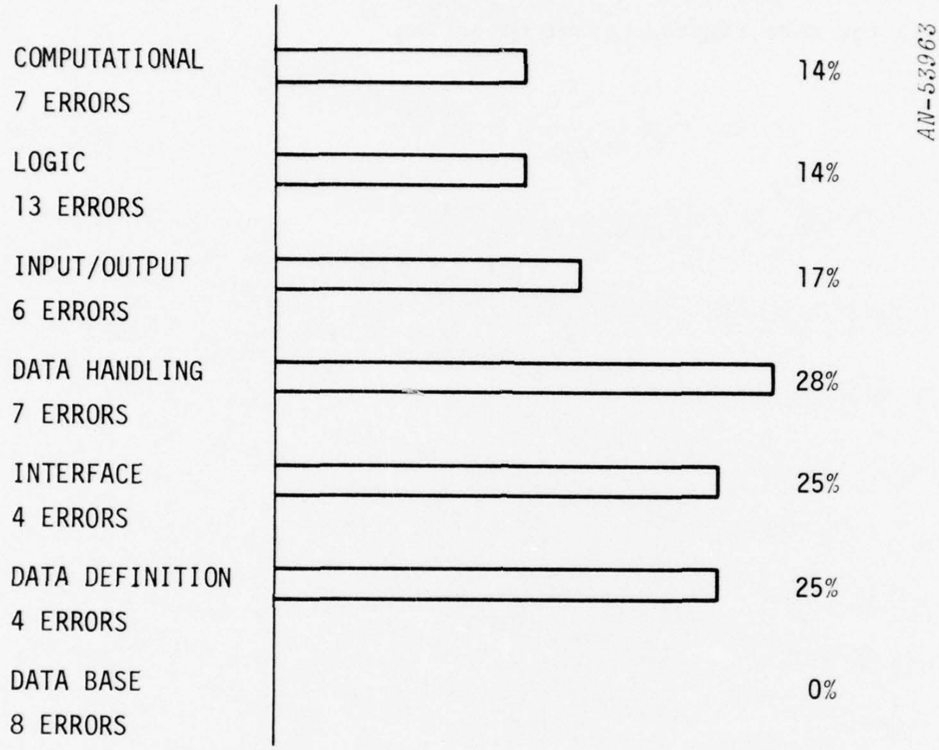


Figure 5.4. Static Analysis: Frequency of Detected Errors by Category

It is our purpose, however, to report the types of errors that can be detected by these techniques. The results of the experiment can also be used as a reference for tool developers seeking to sharpen their tools for more rigorous error detection.

6. MULTI-ERROR EXPERIMENT

A multi-error experiment was conducted to evaluate the utility of static analysis and path testing under more realistic conditions where several errors exist in a program. The experimental conditions were designed to simulate a typical software testing environment in which the program can be compiled and run but the performance or output does not meet specifications.

There are two aspects of the multiple error situation which makes it quite different from the single error conditions. First, the actual number of errors in a program is not known. The tester might try to estimate the number of expected errors, but will not be sure of the extent of the testing task. Testing strategies may be adjusted on this subjective assessment. Also, estimates of the testing time required and the degree of testing completeness will be based on this imperfect information.

The second aspect of multiple errors not found in single error conditions is the problem of one error masking the effects of another. The syndrome of "just one more error" is due at least in part to error symptoms which suddenly appear when an error is corrected. Many times it is difficult to determine whether latent errors are exposed or new errors are introduced when "correcting" a suspected error. There is also a fatigue factor or saturation limit on the number of errors one tester can find, and this limit is almost always less than the actual number of errors in a program.

6.1 DESCRIPTION OF THE MULTI-ERROR EXPERIMENT

The multi-error testing environment was established by seeding the 5000-line FORTRAN test object (program) with 22 of the errors used in the single-error experiment. The error categories and frequency of seeded errors are shown in Fig. 6.1. This collection of errors was the largest set which could be introduced at one time

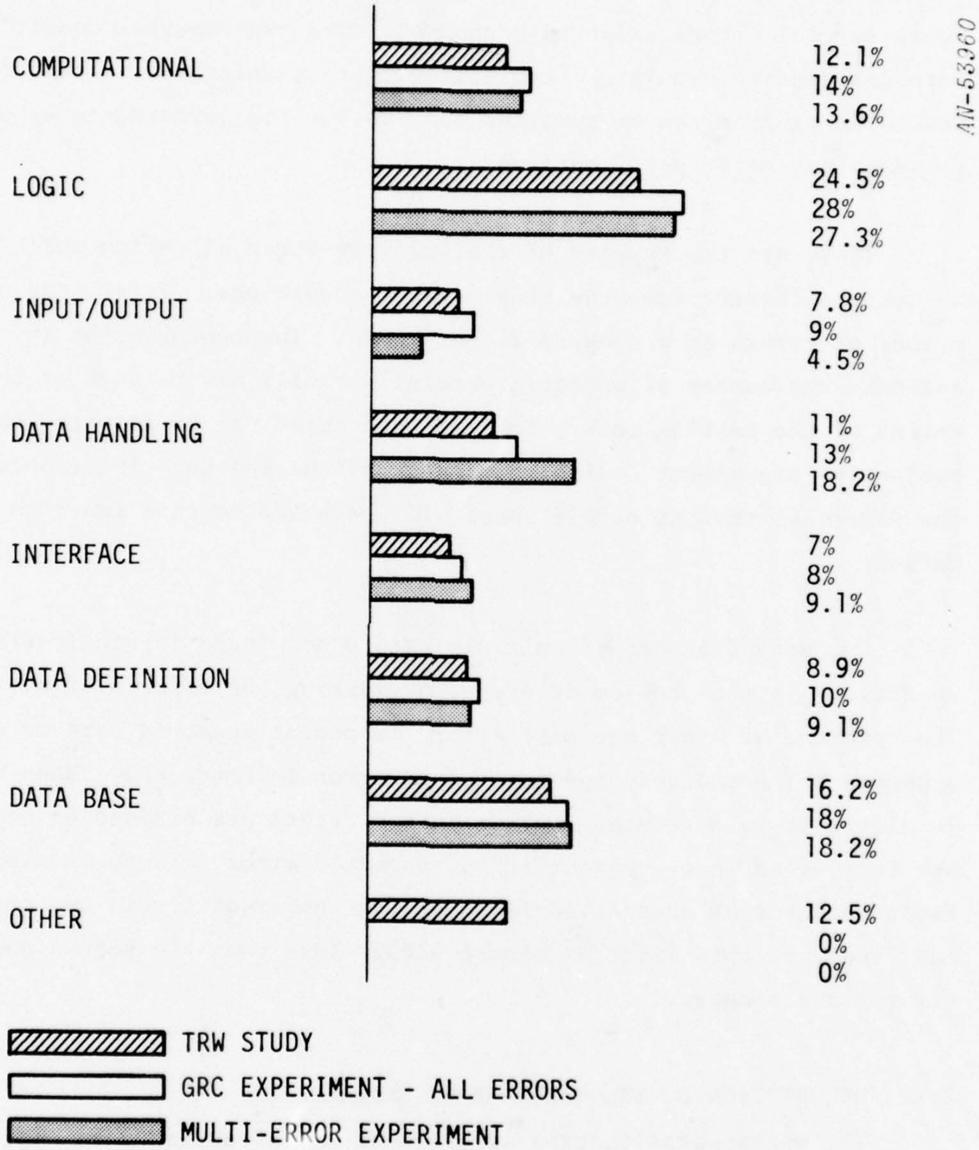


Figure 6.1. Error Frequency in Major Categories

and still have the program run to "normal completion." Figure 6.2 shows a comparison of the number of errors seeded with the number of errors found in other delivered software. This graph, taken from Gannon,[1] indicates that 22 errors could be easily expected in a program of 5000 lines which has been acceptance-tested.

Two testers analyzed the error-seeded program--one using SQLAB for static analysis and path testing and the other using the debugging trace facility provided by the compiler. The two testers worked independently and neither was involved with the single-error experiment. The number of seeded errors was not disclosed to the testers.

Both testers were allowed the same amount of time (120 hours) to conduct their tests. Both worked from the same test object and test dataset, and both used the same computer facility. Both testers were free to use extended compiler reports, insert debugging print statements, and modify the supplied dataset.

Activity reports were prepared as in the single-error experiment to document the error analysis and correction process. A log was also kept to help document the sequence of actions taken in detecting errors.

6.2 RESULTS OF THE MULTI-ERROR EXPERIMENT

The results of the multi-error experiment are difficult to interpret for a number of reasons. Individual performance in programming and debugging is highly variable, and since only two people participated in this phase of the project, statistical measures cannot be derived with confidence. There are, however, some interesting comparisons to be drawn from the data collected and some ideas for improving testing tools and techniques.

---

[1] C. Gannon, "A Verification Case Study," Proceedings of AIAA Computers in Aerospace Conference, Los Angeles, November 1977.

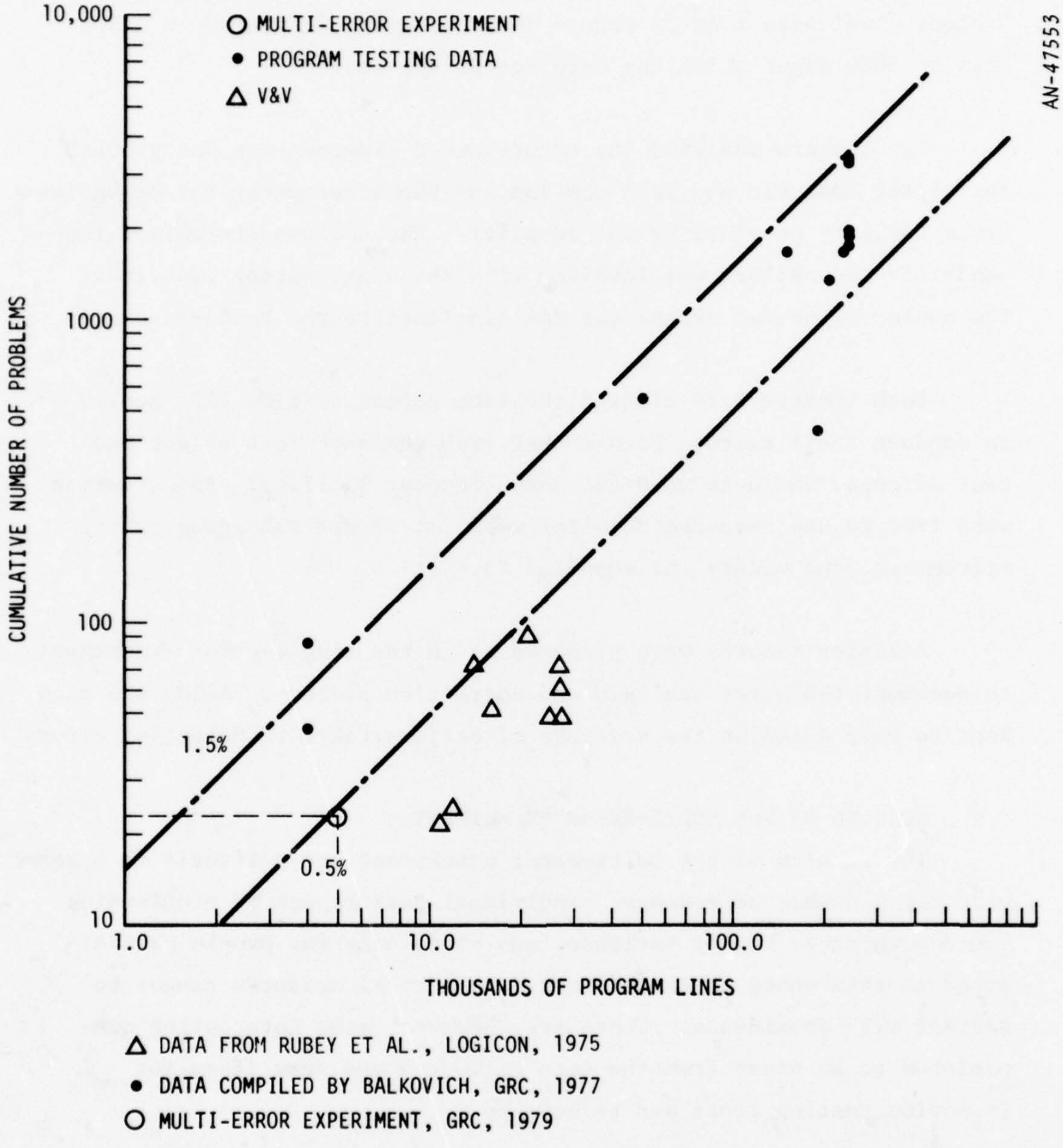


Figure 6.2. Errors in Delivered Software.

The variation in individual performance in programming and debugging was found to range over more than an order of magnitude by Sackman[1] in the early 1960s. More recent experiments by Myers[2] confirm this variability and indicate that modern computer science has not improved this aspect of human fallibility. From these independent results it is surprising how closely the results of the multi-error experiment compare.

6.2.1 Error Detection Results

The results of the multi-error experiment are presented in Table 6.1 which is organized by error category. Of the 22 seeded errors, 11 were found by the tester using the SQLAB test tools and 15 were found by the tester using the debugging trace facilities provided by the compiler. Nine of the errors were found by both testers. The bar graph in Fig. 6.3 provides an overview of the categories of errors detected by each tester.

The information in Table 6.1 is presented in another form in Fig. 6.4, organized by the order in which the errors were discovered by the two testers. The horizontal axis represents the sequence in which errors were found by the tester using the SQLAB test tools. The vertical axis represents the sequence in which errors were found by the tester using the compiler's debug-trace facility. The error numbers and their categories appear at the coordinate positions corresponding to when they were discovered. For example, error E047 was the sixth error found by the SQLAB tester and the eleventh error found by the other tester. Hence, error E047 appears at coordinate position (6,11) in the figure.

---

[1] H. Sackmann, Man-Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing, Petrocelli Books, 1970.

[2] G. J. Myers, "A Controlled Experiment in Program Testing and Code Walkthrough/Inspections," CACM, Vol. 21, No. 9, Sept. 1978.

TABLE 6.1

MULTIPLE ERROR EXPERIMENT - ERRORS FOUND AND RESOURCES EXPENDED

Error Category		Error Number	SQLAB-BASED TESTER				COMPILER-BASED TESTER			
			Found	Sequence	Engr. Time (hours)	Comp. Time (Seconds)	Found	Sequence	Engr. Time (Hours	Comp. Time (Seconds)
Computational	A100	E007	✓	10	3.0	12.7	✓	12	3.5	81.6
		E008								
	A200	E018								
Logic	B200	E013			10.5	51.3			3.5	80.6
	B300	E015	✓	3	1.25	1.7	✓	1	1.0	17.8
		E076	*		4.0	19.3	✓	10	9.2	94.8
	B400	E032	✓	11	1.0	4.4			3.5	80.6
	B500	E001					✓	14	2.2	50.0
		E089					✓	15	4.75	79.8
Data Handling	D200	E047	✓	6	2.0	3.9	✓	11	1.67	10.4
		E069	✓	2	1.0	0.6	✓	3	0.5	10.1
	D400	E036	✓	1	0.5	1.3	✓	5	3.1	73.2
	D900	E002	*		25.5	337.6	✓	6	2.67	25.5
Output	E600	E014	✓	4	3.5	11.4	✓	2	1.0	17.8
Interface	F200	E097					*		.25	20.6
	F700	E028			1.5	4.3	✓	8	2.0	20.7
Data Definition	G100	E085	✓	5	2.0	33.9			2.4	30.8
	G200	E067	✓	7	.75	2.0	✓	7	1.25	24.6
Data Base	H200	E070					✓	13	5.1	71.3
		E072	✓	9	3.25	20.1	✓	4	0.33	10.0
	H300	E009	✓	8	2.0	4.6	✓	9	2.0	40.8
		E078								
TOTALS		22	11		72.	585.	15		50.	792.

✓ Found the error and made appropriate correction.

* Corrected some symptons of the error.

— Detected by Static Analysis.

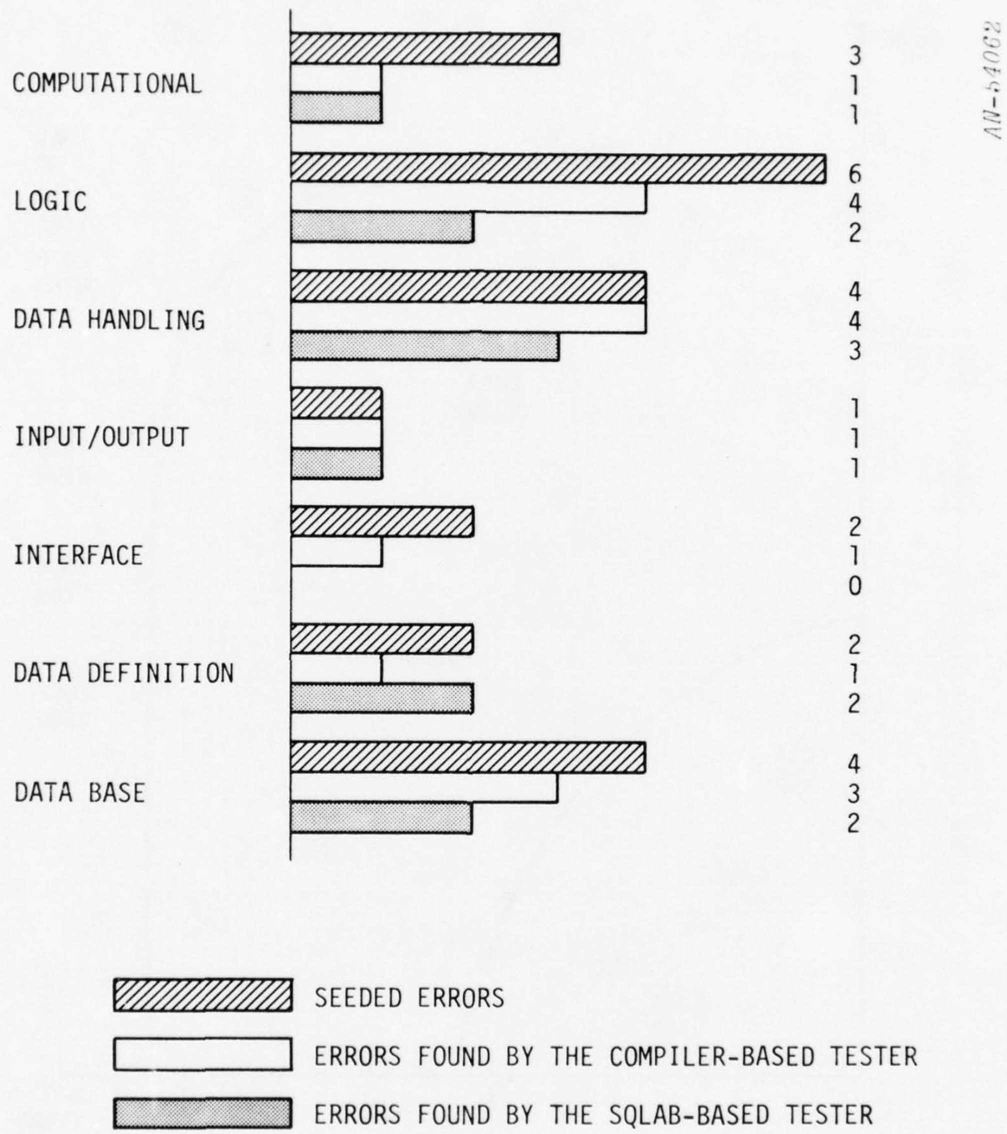
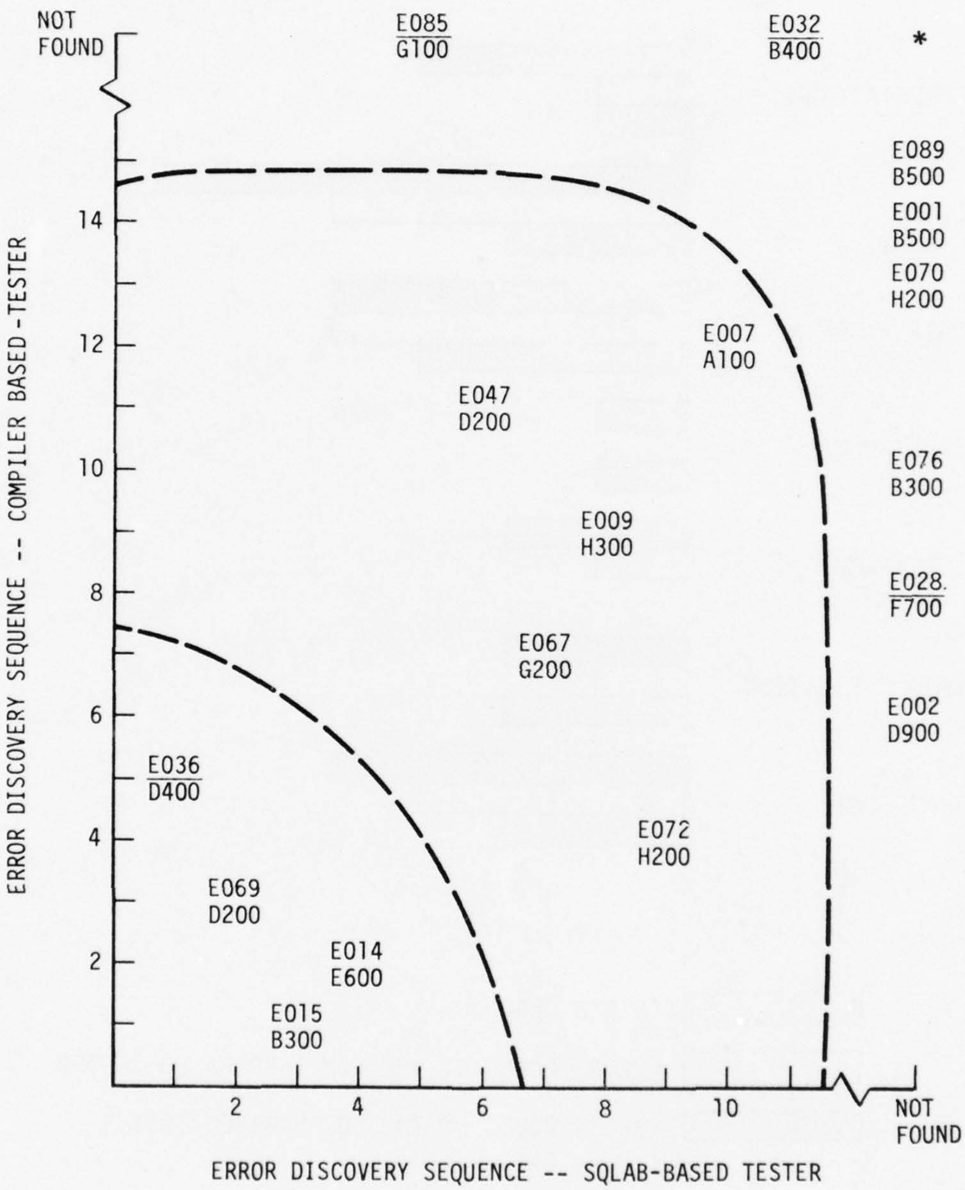


Figure 6.3. Categories of Errors and Method of Detection in the Multi-Error Experiment



*FIVE ERRORS NOT DETECTED BY EITHER TESTER

Figure 6.4. Order of Error Discovery in Multi-Error Experiment

Several direct observations can be made from the representation of the data in Fig. 6.4 which were not apparent in Table 6.1. The four errors included in this experiment which were found using SQLAB's static analysis capability and discussed in Sec. 5.3 are underscored in the figure. Error number E036 was an easy one, found early by both testers. This was a data handling error (category D400) in which the wrong variable name was used as an argument in a subroutine call. The other three errors seemed to be much more elusive.

The two errors found by the SQLAB tester but not by the other tester were diagnosed by static analysis. The first was a data definition error (category G100) which was caused by changing the name of a variable. The change in the name caused a change in its default datatype and, hence, its attributes. This error, number E085, was indicated by two mode warnings which were camouflaged by 17 other innocuous mode warnings in the containing routine.

The second error, number E032, was introduced by deleting a conditional branch statement from a module, simulating a "missing logic" error (category B400). This error was clearly diagnosed as making a section of code unreachable.

Error number E028 was found by the tester using the compiler's trace facility but not by the SQLAB tester. This error was an interface error between program modules (category F700). It was introduced by simply reversing the order of two parameters in a subroutine heading. SQLAB generated 12 mode warnings about this error but none of the warnings was even near the source of the error. The warnings were reported at the CALL statements which invoked the erroneous subroutine but none appeared at the source of the trouble. The problem was compounded because the warnings all disappeared when the offending subroutine (which had no reported errors) was removed from the analysis to expedite testing.

Four distinct classes of errors can be derived from the data in Fig. 6.4. The four errors clustered in the lower lefthand corner represent easy errors which are quickly and easily diagnosed and hence are perhaps not serious problems. The other five errors found by both testers are somewhat more difficult to find and hence might be classified as moderately difficult. The collection of eight errors found by one tester and not the other form a class of errors which are more difficult to diagnose than the errors found by both testers. The last five errors, which were not discovered by either tester, represent a class of subtle errors which are likely to escape detection during formal testing.

6.2.2 Resources Expended

The resources in terms of engineering and computer time used by the two multi-error testers are presented in Table 6.1. Only the times which could be directly attributed to individual errors are recorded in this table. Hence, some of the entries have been left blank. Also, the total times reported are larger than the sums of the individual times recorded in each column.

The first item to note is the total engineering time spent by the two testers. Each tester was allotted 120 hours for their task. The SQLAB-based tester spent 72 hours; the compiler-based tester spent only 50. Both testers expressed a feeling of having reached the limit of their effectiveness in finding more errors. The SQLAB-based tester seemed overwhelmed by the complexity of the mathematics and the inscrutability of the program. The lack of specifications for the program and documentation from earlier testing efforts also contributed. The compiler-based tester thought there was probably only one error left in the program (when in fact there were seven more errors) but felt it would take an inordinate amount of time to diagnose.

Perhaps too much emphasis has been placed on the testing tools and not enough on human factors. The psychological stress of testing and debugging a program can be severe. Both testers found the task quite difficult and frustrating. The satisfaction of finding an error did not seem sufficiently rewarding to stimulate renewed efforts. The reward was often the exposure of symptoms of more errors.

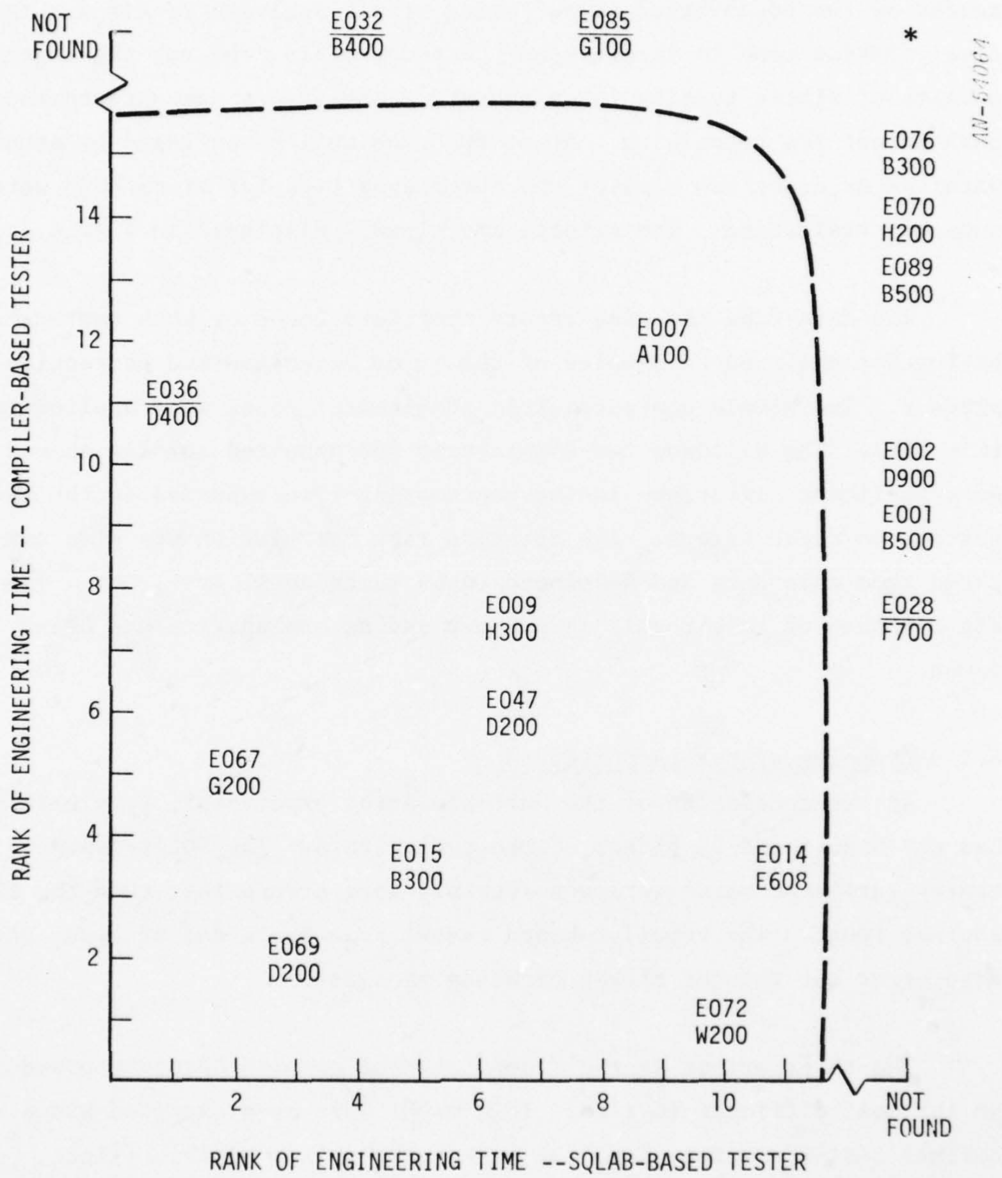
Comparison of the resources (engineering time, computer time, etc.) used by each of the testers shows no statistically significant differences based on Sackmann's and Myers' evidence of individual variability. The time spent per error which can be derived from the measured data showed the largest difference between the two testers. The tester using the SQLAB test tools spent 72 hours and found 11 errors or about 6.5 hours per error. The tester using the compiler's trace facility spent 50 hours and found 15 errors or about 3.3 hours per error. The ratio of 6.5:3.3 (1.97), however, is still not statistically significant. The compiler-based tester felt that the debugging-trace facility reduced the time he spent to about one-half of the time he would have spent inserting debugging print statements manually.

Another parameter derived from the measured data was the amount of time spent per computer run. The tester using the SQLAB test tools spent 72 hours and ran 78 jobs or about 56 minutes per run. The tester using the compiler's trace facility spent 50 hours and ran 90 jobs or about 33 minutes per run. The ratio of 56:33 (1.70) compares closely with the ratio of 1.97 found for the time spent per error. The tester using the SQLAB test tools observed that many of SQLAB's diagnostics and warnings indicated violations of programming standards which did not affect the computation and hence were not counted as errors. Each warning had to be checked out, however, which may account for some of the differences in performance.

The activity reports which were prepared by both testers indicated that the tester using the compiler's debugging facilities was better able to discern the effects due to different errors in the program. He was, therefore, able to isolate problems, focus his attention, and find errors more quickly. His approach was to work on finding the cause of the first discrepancy which appeared in the output. The rest of the output was disregarded because it contained symptoms of other errors which would not help locate the first error.

The other tester, using the SQLAB test tools, spent a considerable amount of time studying the reports generated by the tools and checking out the reported errors and warnings. The test program contained many violations of modern programming standards and practices which SQLAB faithfully reported. Only four of the 22 seeded errors were found by SQLAB's static analysis, yet the static analysis reports contain 51 unrelated error and warning messages. Most of the warnings were mixed-mode Holerith expressions, and the error messages flagged "uninitialized" variables that had been set via their equivalenced names. This aspect of SQLAB's reports indicates the importance of using them during program development to enforce good programming practices. The tester was also misled by a modification which cleared several error symptoms but did not correct the error. The modification created a more subtle "double error" in a section of the program which was thought to be working correctly.

The engineering time spent by the two multi-error testers is presented in another form in Fig. 6.5. In this figure the horizontal axis represents the time spent by the SQLAB based tester and the vertical axis represents the time spent by the compiler based tester. The scale represents the rank order of the engineering times recorded in Table 6.1. Errors which required more time have higher rank.



* ALL ERRORS NOT FOUND BY EITHER TESTER

Figure 6.5. Multi-Error Experiment Engineering Time Resources

The first observation which one can make is that the errors detected by the SQLAB-based tester using static analysis required relatively little time to identify and correct. This confirms the expected utility of static testing for a subset of the errors encountered and is perhaps not too surprising. Error E085 was well-camouflaged by other warnings as described earlier and the diagnostics for error E028 were somewhat misleading. The effects are clearly displayed in Fig. 6.5.

The data from the nine errors that were found by both testers can be further analyzed as samples of the error detection and correction process. Two simple non-parametric statistical tests were applied to this data. The Wilcoxon two-sample test for unpaired samples showed no significant difference in the engineering time expended by the two testers on these errors. The Spearman rank correlation was also computed from this data and was found to be quite small (r=-.208). The significance of this result is unknown and no explanation has been found.

6.2.3 Examples of Errors Not Found

At the conclusion of the multiple error experiment, five errors had not been found by either of the two testers. The SQLAB-based tester estimated there were considerably more errors left than the 11 she had found. The compiler-based tester knew there was at least one more error but thought it was probably the last.

The three errors in the "computational errors" category proved to be the most difficult to find. This might have been expected since neither tester was very familiar with formulas for missile flight, elliptic orbits, or coordinate transformations in three dimensions. Error number E007 was the only error in this category found by both testers. It was one of the last errors found and required more than average time to discover.

Error number E008 was very similar in form to error E007 but was not found by either tester. For error E008 an intermediate result in the calculation of the Euler angles for an orbit was calculated using the wrong operand in the equation [(cos(β) instead of sin(β)]. A major contributing factor to the difficulty with this error was that the correct values for the computed Euler angles were not available to the testers. Only after many steps of intervening computations were the effects of this error finally exposed.

Unit testing of the module containing error E008 would have readily shown the existence of an error. It is believed, however, that this error would have been difficult to isolate and correct even if the search was restricted to the program module containing the error.

Error number E018 was the third error in the "computational" category and represented the sub-category "Incorrect use of parenthesis" (A200). The calculation of the length of the major axis of an elliptic orbit was changed from

```
A=GCON*ADIV(R,(2.*GCON-R*(VR**2+VQ**2)))
```

to

```
A=GCON*ADIV(R,(2.*GCON-R*(VR+VQ)**2))
```

Several additional orbital parameters were then computed using the value of A. As with error E008, the correct values of the orbital parameters were not available to the testers and the effects were not evident until some time later. It should be noted that the tester in the single-error experiment also failed to find this error.

Only one error in the "logic error" category was missed by both testers. This was error number E013 in which the wrong statement label was assigned to a program variable thus causing a control flow error. The "assigned GOTO" is one of FORTRAN's more baroque features and it was used extensively in the module containing this error. In fact, the control flow in this module was so complex that SQLAB's restructuring capability failed to sort it out. SQLAB's restructuring capability is used to convert unstructured code into structured code automatically but in this case it failed to complete the analysis of all the possible paths within this module.

Error number E078 was intended to simulate a database error, "data units incorrect" (H300). The seeded error also looks like an "incorrect operand in equation" (A100), a computational error, although it does exhibit units problems. In the calculation of the position, velocity, and acceleration of an object in orbit the intermediate result

```
Q=2.*ATAN2(X1,X2)
```

was changed to

```
Q=TWOPI*ATAN2(X1,X2)
```

where TWOPI was a variable initialized to 6.2832 (radians). The function ATAN2 (arctangent) returns an angle also with units of radians. Hence the value computed for Q, which is an angular displacement, would have incorrect units of radians-squared. Neither of the multiple error testers discovered this error. The single error tester found the offending statement but was unable to synthesize the correction.

7 CONCLUSIONS

This project provided the opportunity for a critical and objective assessment of the only two automated testing techniques that are mature enough to be useful, path testing and static analysis. There are two unique aspects of this project that distinguish the results from similar software testing evaluation experiments.

1. The test engineers did not know the type or location of the program errors.
2. An automated test tool was used for error detection.

Experience has shown us that a simulated tool evaluation of a particular testing technique based on knowing the type and location of an error does not address many of the difficulties faced by using a real tool and not knowing anything about the error(s). Because software normally contains numerous peculiarities of design or implementation, what constitutes an error may not be obvious. Furthermore, automated test tools (like compilers) are unforgiving in their consistence checking. Static analysis is particularly affected by this characteristic. For a single, erroneous mixed mode expression, there may be hundreds of correct, intentional ones, yet the static analyzer will faithfully report all inconsistencies.

Similarly, while executing a particular path might cause an error to manifest itself in the output, doing so may cause many other paths to be executed, perhaps completely masking the error. This problem becomes acute when the output-producing code is distant from the source of error. If the error location is known from the start, it may be a simple matter to determine the effectiveness of a particular testing technique.

While the individual characteristics of the test tool used in the experiments undoubtedly played a part in the results, the primary testing effectiveness, we feel, is due to the two techniques used.

For example, the DAVE system, a static analyzer, was found to detect one class of errors (too many/too few statements in a loop: B500) but unable to detect others (such as missing logic or condition tests: B400). Similarly, the path testing tool used in the experiments, SQLAB, did not provide the valuable dynamic tracing information provided by other path-testing tools such as the JOVIAL Automated Verification System (JAVS).[1] However, we believe that the data generated in the experiments provide a good foundation for some conclusions about the testing methods.

As described in earlier sections, for these experiments an error is incorrect implementation of a specification or reliance on a compiler's, operating system's, or machine's nonstandard capability. Examples of "nonstandard" capabilities are assuming storage is preset to zero or assuming arrays adjacently declared necessarily share contiguous storage space. The "nonstandard" type of errors were removed from the test object before starting the experiment, in order to not make the test analyst's task of finding seeded errors even harder. This removal did not, however, eliminate the error and warning messages described in Sec. 6.2.2.

In addition, errors derived during the error-seeding process that, though the site was executed, did not manifest themselves in the output, were not sown in the program for subsequent detection. This was done because, owing to the lack of program specification, a listing of the correct program's output was used as the only specification. Although 22 errors (25% of the total errors generated by the error-seeding process) whose sites were executed were not used during the

---

[1] C. Gannon and N. B. Brooks, JAVS Technical Report, Vol. 1: User's Guide, General Research Corporation CR-1-722/1, June 1978.

experiments, their existence is the basis for one major conclusion of this evaluation: Errors may reside on paths and statements that, although executed, may not show up in the output for the test data used. Thus testing must face the issue that more information must be supplied in a program during development (at, undoubtedly, greater programmer effort) to (1) direct testing of legal sequence of paths, and (2) specify functional correctness of statements and paths.

7.1 EFFECTIVENESS FOR ERROR DETECTION

When an error is known to exist, as in the error-type detection experiment (Phase 2--single-error experiment), it was found that 40 percent of the errors were readily found by inspection, 45 percent more were found using path-testing assistance, and the remaining 15 percent were not found or were improperly corrected. The errors found using path testing were significantly more difficult than those found by inspection, although no quantitative measure can be given for "difficulty." The average time spent on errors found by inspection was one hour, whereas for the more difficult errors found by path testing the average time was three hours.

Path-testing tools do not generate error messages indicating the source of an error in a program. They do, however, provide a great deal of assistance by narrowing the scope of the search for errors and reducing the number of possible error sites which must be investigated. Hence, path testing is really enhanced inspection. The enhancement increases the probability of finding an error by inspection from 40 to 80 percent.

Path sequence information was found (by the tracing capability of the compiler used in Phase 3--multi-error experiment to be more valuable for finding errors than path coverage information. The major drawback of typical path tracing techniques is the volume of rather useless output surrounding usually one or two lines indicating incorrect behavior. An improvement which could be supported by a test tool would be a condensed report which would retain the valuable sequence information. We feel that research should be directed toward determining what a "valuable" sequence is. Of special consideration are sequences which are functionally important and those which lead up to or include threshold or boundary conditions.

Path testing was found to be helpful in all error categories. There are examples of errors in each category which required the use of path coverage information to discover the source of the trouble. However, seven of the nine errors not found in the error type detection experiment were from the "computational," "logic," and "database" categories, indicating some weakness of path testing in these areas.

Static analysis is credited with finding nine of the 49 errors used in the error-type detection experiment--one of which was not found by the path testing analyst. The economy of static analysis is shown by the cost of its use (two engineering hours and 24 computer seconds) compared with the path testing cost for the same errors of 13.5 engineering hours and 110 computer seconds. Even though only one of the errors generally more difficult to diagnose was found using static analysis, it is an effective tool for screening some errors. It has the advantage of generating diagnostic messages about errors at their statement location, and it does not depend on any knowledge of error manifestation.

7.2 EFFECTIVENESS FOR VERIFICATION

Path testing provides little support for determining the correctness of programs, even through exhaustive path coverage. The correct functioning of a program has to be checked by other means. The primary function provided by path coverage is an indication of parts of a program which have not been exercised. Full path coverage does not ensure complete or sufficient testing, since errors may occur on sequences of paths which have not been tested. Furthermore, path testing and static analysis are not capable of evaluating functional correctness unless test data are derived from the software specification.

Even with these limitations in mind, there appears to be considerable room for improvement in path-oriented verification tools. The missing ingredient seems to be a specification of the legal path sequences which a program should be allowed to traverse. The combinatorial nature of this problem makes it intractable for even small programs. Approximations or heuristic algorithms, however, may yield acceptable solutions for many real programs.

Hamlet[1] describes a promising approach of using "computational specifications" to complement the standard use of "functional specifications" for programs. Computational specifications impose additional constraints on how results are to be obtained. Functional testing can be performed only on a small subset of the input domain. However, if correct results are obtained using the prescribed computation, then the small sample tests can be shown to be reliable. We expect that path sequence information will be a major component in such computational specifications.

7.3 VALUE OF ERROR SEEDING

The primary advantage of seeding errors for experiments is the control it provides over the types and distribution of errors in a program. Programs with authentic errors which satisfy requirements for testing experimental hypotheses are simply not available on demand. This control, we feel, is more important than the true authenticity of the errors.

---

[1] R. G. Hamlet, "Critique of Reliability Theory," Workshop Digest, Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, December 1978.

The three testers involved in the error type detection and testing technique evaluation experiments in this study agreed that the seeded errors were very realistic. They did not feel that the environment was at all artificial or contrived. This was probably due to the care taken in error selection and seeding. It also indicates that the results of the experiments apply directly to real programs with authentic errors.

One of the factors that was not controlled in our experiments was the <u>subtlety</u> of the seeded errors and, hence, the difficulty of the discovery. Defining subtlety may not be easy. In general, the most difficult errors to discover were those which propagated incorrect results through long sequences of computations with no outward sign of trouble. When the symptom finally surfaced, the link back to the originating error was completely obscured. Using degree of obscurity as a measure of subtlety, one could construct a test program seeded with easy errors, difficult errors, or some combination to test an hypotheses about the effectiveness of a particular test tool or method.

An analogy can be drawn between testing software and other scientific investigations. Error-seeding experiments correspond to laboratory experiments where conditions can be controlled and many parameters can be measured. Production programs in actual use correspond to field studies where the conditions cannot be controlled and some measurements cannot be made. The analogy extends to the need for relevancy between error-seeding experiments and delivered software just as the need exists for relevance between laboratory experiments and field studies. We highly recommend the practice of error-seeding to software testing and verification tool developers as a measure of effectiveness.

APPENDIX A

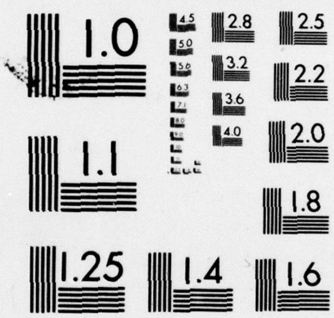
Small Programs for

Preliminary Analysis

AD-A071 050 GENERAL RESEARCH CORP SANTA BARBARA CA SYSTEMS TECHNO--ETC F/G 9/2
AN EXPERIMENTAL EVALUATION OF SOFTWARE TESTING.(U)
MAY 79 C GANNON, R N MEESON, N B BROOKS F49620-78-C-0103
UNCLASSIFIED GRC-CR-1-854 AFOSR-TR-79-0733 NL

2 OF 2
AD
A071050

END
DATE
FILMED
8-79
DDC


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
      PROGRAM  SINEFCN (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT,TAPE12)
C
C     DRIVER PROGRAM TO TEST THE DOUBLE PRECISION SINE FUNCTION
C     REG MEESON      7/11/78
C
      DOUBLE PRECISION   SIN, DSIN, DBLE, REF, VAL, E
      REAL    X
C
      WRITE(6,100)
   10 READ (5,110) X, E
      WRITE(6,120) X, E
      IF( E .EQ. 0. ) STOP
      REF = DSIN( DBLE(X) )
      VAL = SIN(X,E)
      WRITE(6,130) REF, VAL
      GOTO 10
C
  100 FORMAT( 26H SINE FUNCTION TEST DRIVER // )
  110 FORMAT( F10.4, D10.2 )
  120 FORMAT( 3H X=, F10.4, 7H     E=, D20.12 )
  130 FORMAT( 1H+, 45X, 4HREF=, D20.12, 9H      VAL=, D20.12 )
C
      END
      DOUBLE PRECISION FUNCTION SIN(X,E)
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 77.
C
C     THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
      DOUBLE PRECISION E,TERM,SUM
      REAL X
      TERM=X
      DO 20 I=3,100,2
      TERM=TERM*X**2/(I*(I-1))
      IF(TERM.LT.E)GO TO 30
      SUM=SUM+(-1**(I/2))*TERM
   20 CONTINUE
   30 SIN=SUM
      RETURN
      END
*
  .5236     1.000D-08
 3.14159    1.000D-08
 -.1        1.000D-08
 0.         0.000D+00
```

```
      PROGRAM  CURRENT (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT,TAPE12)
C
C     CURRENT COMPUTING PROGRAM
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 79.
C
C     INPUT VALUES FOR RESISTANCE, FREQUENCY AND INDUCTANCE
      READ(5,20) R,F,L
   20 FORMAT(3F10.4)
C     PRINT VALUES OF RESISTANCE, FREQUENCY AND INDUCTANCE
      WRITE(6,30) R,F,L
   30 FORMAT(3H1R=,F14.4,4H  F=,F14.4,4H  L=,F14.4)
C     INPUT STARTING AND TERMINATING VALUES OF CAPACITANCE,AND INCREMENT
```

```
      READ(5,40) SC,TC,CI
   40 FORMAT(3F10.6)
C     SET CAPACITANCE TO STARTING VALUE
      C=SC
C     SET VOLTAGE TO STARTING VALUE
      V=1.0
C     PRINT VALUE OF VOLTAGE
   50 WRITE(6,60) V
   60 FORMAT(3H0V=,F5.0)
C     COMPUTE CURRENT AI
   70 AI = E / SQRT(R**2 + (6.2832*F*L - 1.0/(6.2832*F*C))**2)
C     PRINT VALUES OF CAPACITANCE AND CURRENT
      WRITE(6,80) C,AI
   80 FORMAT(3H0C=,F7.5,4H  I=,F7.5)
C     INCREASE VALUE OF CAPACITANCE
      C = C + CI
      IF (C .LE. TC) GO TO 70
C     INCREASE VALUE OF VOLTAGE
      V = V + 1.0
C     STOP IF VOLTAGE IS GREATER THAN 3.0
      IF (V .LE. 3.0) GO TO 50
      STOP
      END
-
  10.       .159      10.
  .08       .12       .01
```

```
      PROGRAM  NUMALPH (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT,TAPE12)
C
C     A PROGRAM WITH A SUBTLE INITIALIZATION ERROR
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 80.
C
C     AUGMENTED TO PRODUCE SOME OUTPUT      7/11/78     REG MEESON
C
      DIMENSION NUM(80),NALPHA(80)
      DATA NBLANK /1H /
      READ (5,101) NALPHA,NUM
  101 FORMAT (80A1,T1,80I1)
      WRITE(6,102) NALPHA, NUM
  102 FORMAT( 11H INPUT DATA / 1H0,80A1 / 1H ,80I1 )
      NUM = 0
      N = 0
      DO 30 I = 1,80
      IF (NALPHA(I) .EQ. NBLANK) GO TO 30
      N = N + 1
      NSUM = NSUM + NUM(I)
   30 CONTINUE
      WRITE(6,103) N, NSUM
  103 FORMAT( 30H0THE NUMBER OF DIGITS FOUND IS, I3 /
     $        29H AND THE SUM OF THE DIGITS IS, I4 )
      STOP
      END
-
3   55   127   3467   124689
12345     1 3 5 7 9     2 4 6 8 10 12 14 16 18 20     5 10 15 20 25 3
```

```
      PROGRAM  BALANCE (INPUT,CUTPUT,TAPE5=INPUT,TAPE6=OUTPUT,TAPE12)
C
C     COMPUTES A TABLE OF MONTHLY BALANCES AND INTEREST CHARGES FOR
C     A GIVEN PRINCIPAL AMOUNT, INTEREST RATE, AND MONTHLY PAYMENT.
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PRGRAMMING STYLE
C               PAGE 85.
C
C     CONVERTED TO FORTRAN      7/11/78      REG MEESON
C
      REAL   A, R, M, B, C, P
C
   10 READ (5,101) A, R, M
  101 FORMAT(3F10.4)
      WRITE(6,102) A, R, M
  102 FORMAT(14H THE AMOUNT IS,F10.2,
     $      23H   THE INTEREST RATE IS,F6.2,
     $      25H   THE MONTHLY PAYMENT IS,F8.2)
      IF (M .LE. A*R/1200.) GO TO 30
      WRITE(6,103)
  103 FORMAT(1H-,
     $59H          MONTH        BALANCE    CHARGE         PAID ON PRINCIPAL / )
      B=A
      DO 18 I=1,60
      C=B*R/1200.
      IF (B+C .LT. M) GO TO 20
      P=M-C
      B=B-P
   18 WRITE(6,181) I, B, C, P
  181 FORMAT (I13, 3F13.2)
   20 BPLUSC = B+C
      WRITE(6,201) BPLUSC
  201 FORMAT (35H0THERE WILL BE A LAST PAYMENT OF    , F8.2)
      GO TO 10
   30 WRITE(6,301)
  301 FORMAT (30H0UNACCEPTABLE MONTHLY PAYMENT )
      GO TO 10
      END
-
  500.       18.        45.
  100.        9.        17.
 1200.       15.        12.
```

```
      PROGRAM  BINSRCH (INPUT,OUTPUT,TAPE12)
C
C     BINARY SEARCH PROCEDURE TO FIND AN ELEMENT *A* IN A TABLE *X*
C     THE ELEMENTS IN *X* MUST ALREADY BE SORTED INTO INCREASING ORDER
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 87.
C
      DIMENSION X(200),Y(200)
      READ 50, N
   50 FORMAT(I5)
    2 READ 51, (X(K), Y(K), K = 1, N)
   51 FORMAT (2F10.5)
      READ 52,A
   52 FORMAT (F10.5)
      IF (X(1)-A)41, 41, 11
   41 IF(A-X(N))5, 5, 11
   11 PRINT 53,A
```

```
53 FORMAT(1H ,F10.5,
  1      26H IS NOT IN RANGE OF TABLE.)
   STOP
 5 LOW = 1
   IHIGH = N
 6 IF (IHIGH-LOW-1)7, 12, 7
12 PRINT 54, XLOW, YLOW, A, XHIGH, YHIGH
54 FORMAT(1H 5F10.5)
   STOP
 7 MID = (LOW + IHIGH)/2
   IF (A-X(MID))9, 9, 10
 9 IHIGH = MID
   GO TO 6
10 LOW = MID
   GO TO 6
   END

   7
 -3.2       1.
  -.1       2.
  1.3       3.
  8.7       4.
 20.5       5.
 22.8       6.
697.4       7.
   1.
```

```
      PROGRAM  INTEGR8 (OUTPUT,TAPE2=OUTPUT,TAPE12)
C
C     INTEGRATES A POLYNOMIAL BY TRAPEZOIDAL APPROXIMATION
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 91.
C
      AREA=0.
      X = 1.
      DELTX=0.1
 9    Y=X**2+2.*X+3.
      X=X+DELTX
      YPLUS=X**2+2.*X+3.
10    AREA=AREA+(YPLUS+Y)/2.*DELTX
      IF(X-10.)9,15,15
15    WRITE(2,7)AREA
7     FORMAT(E20.8)
      STOP
      END
```

```
      PROGRAM  FLOATPT (INPUT,OUTPUT,TAPE1=OUTPUT,
     $                  TAPE2=INPUT,TAPE3=OUTPUT,TAPE12)
C
C     TESTS FOR EXACT EQUALITY BETWEEN COMPUTED FLOATING POINT NUMBERS
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 93.
C
C     RIGHT TRIANGLES
      LOGICAL RIGHT, DATA
      DO 1 K = 1,100
```

```
      READ (2,10) A, B, C
C     CHECK FOR NEGATIVE OR ZERO DATA
      DATA = A.GT.0. .AND. B.GT.0. .AND. C.GT.0.
      IF(.NOT.DATA) GO TO 2
C     CHECK FOR RIGHT TRIANGLE CONDITION
      A = A**2
      B = B**2
      C = C**2
      RIGHT = A.EQ.B+C .OR. B.EQ.A+C .OR. C.EQ.A+B
1     WRITE(3,11) K, RIGHT
      CALL EXIT
C   ERROR MESSAGE
2     WRITE(1,12)
      STOP
10    FORMAT(3F10.4)
11    FORMAT(I6,L12)
12    FORMAT(11H DATA ERROR)
      END
•
   1.          2.          3.
   5.          12.         13.
   3.          4.          5.
   .05         .12         .13
   .3          .4          .5
   0.          0.          0.
```

```
      PROGRAM  AREATRY (INPUT,OUTPUT,TAPE2=INPUT,TAPE3=OUTPUT,TAPE12)
C
C      FIRST ATTEMPT FOR APPROXIMATING AREA UNDER A CURVE
C
C     SOURCE=   KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 96.
C
    1 AREA=0.0
      READ(2,10)T
   10 FORMAT(F10.4)
      H=0.1
      X=0.0
    2 XN=-X
      AREA=AREA+(6.0*(2.0**XN)+6.0*(2.0**(XN-H)))*0.1/2.0
      X=X+H
      IF(X-T)2,8,9
    8 WRITE(3,33)AREA
   33 FORMAT(7H AREA =,F8.5)
      GO TO 1
    9 CALL EXIT
      END
•
   3.
   5.
   1.
•
```

APPENDIX B

Chronological List of Submitted Papers

The following collection of abstracts, papers and documents was supported by AFOSR F49620-78-C-0103.

1. C. Gannon and R. N. Meeson, "An Empirical Evaluation of Static Analysis and Path Testing," abstract submitted (Jan. 1978) to the Computers in Aerospace Conference II in Los Angeles, California, October 1979.

2. C. Gannon, Empirical Results of Static Analysis and Path Testing of Small Programs, General Research Corporation RM-2225, March 1979.

3. C. Gannon, "Error Detection Using Path Testing and Static Analysis," paper submitted to Computer magazine of the IEEE Society (March 1979).

4. C. Gannon, N. B. Brooks, and R. N. Meeson, An Experimental Evaluation of Software Testing, Final Report, General Research Corporation CR-1-854, May 1979.

5. C. Gannon and R. N. Meeson, "Implications for Test Tool Improvement," to be submitted to COMPSAC 79, the IEEE Computer Society's Third International Computer Software and Applications Conference, Chicago, 1979.

APPENDIX C

Personnel Associated with the Project

The following personnel were contributors to the research effort and set of experiments:

1. Dorothy Andrews, MSEE, University of California, Santa Barbara

2. Jeoffrey P. Benson, PhD, University of Californis, Santa Barbara

3. Nancy B. Brooks, MS, University of Illinois

4. Carolyn Gannon, MSEE, University of California, Santa Barbara

5. Reginald N. Meeson, MSEE, PhD candidate, University of California, Santa Barbara

6. Sabina H. Saib, PhD, University of California, Los Angeles

Bibliography

1. B. W. Boehm, "Software Engineering: R & D Trends and Defense Needs," Proceedings of the Conference on Research Directions in Software Technology, October 1977, cited on p. 1-2.

2. D. J. Reiffer and R. L. Ettenger, "Test Tools: Are They a Cure-All?" Proceedings of the 1975 Annual Reliability and Maintenance Symposium, IEEE 75CHO918-3ROC, January 1975, cited on p. 1-2.

3. J. B. Goodenough, "A Survey of Program Testing Issues," Proceedings of the Conference on Research Directions in Software Technology, October 1977, cited on pp. 1-2, 1-3.

4. W. C. Hetzel, An Experimental Analysis of Program Verification Methods, Thesis, University of North Carolina, Chapel Hill, N. C., 1976, cited on p. 1-3.

5. C. Gannon, "A Verification Case Study," Proceedings of AIAA Computers in Aerospace Conference, Los Angeles, November 1977, cited on pp. 1-3, 6-3.

6. W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," Computer Science Technical Report II, University of California, San Diego, May 1976, cited on pp. 1-3, 1-4.

7. W. E. Howden, "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978, cited on p. 1-3.

8. E. R. Mangold, "Software Error Analysis and Software Policy Implications," IEEE EASCON, 1974, pp. 123-127, cited on p. 1-3.

9. B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974, cited on pp. 1-4, 2-1.

10. R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, April 1978, cited on p. 1-4.

11. D. M. Andrews and J. P. Benson, Software Quality Laboratory User's Manual, General Research Corporation CR-4-770, May 1978, cited on p. 1-5.

12. L. D. Fosdick and C. Miesse, The DAVE System User's Manual, University of Colorado, CU-CS-106-77, March 1977, cited on p. 1-5.

13. T. Plambeck, The Compleat Traidsman, General Research Corporation, IM 711/2, September 1969, cited on p. 3-1.

14. T. A. Thayer, et al., Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, Redondo Beach, California, August 1976, cited on p. 4-1.

15. M. J. Fries, Software Error Data Acquisition, Boeing Aerospace Company, RADC-TR-77-130, Seattle, Washington, April 1977, cited on p. 4-1.

Bibliography, cont.

16. Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974, cited on p. 4-1.

17. H. Sackmann, Man-Computer Problem Solving: Experimental Evaluation of the Time-Sharing and Batch Processing, Petrocelli Books, 1978, cited on p. 6-5.

18. G. J. Myers, "A Controlled Experiment in Program Testing and Code Walkthrough/Inspections," CACM, Vol. 21, No. 9, Sept. 1978, cited on p. 6-5.

19. C. Gannon and N. B. Brooks, JAVS Technical Report, Vol 1: User's Guide, General Research Corporation CR-1-722/1, June 1978, cited on pp, 3-6, 7-2.

20. R. G. Hamlet, "Critique of Reliability Theory," Workshop Digest, Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, December 1978, cited on p. 7-5.